Developer Guide, Version 2

AWS IoT Greengrass



Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

AWS IoT Greengrass: Developer Guide, Version 2

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

What is AWS IoT Greengrass? 1
New features 1
For first-time users 2
For existing users 2
How AWS IoT Greengrass works 2
Key concepts 3
Features of AWS IoT Greengrass 5
Greengrass feature compatibility 7
Choosing your AWS IoT Greengrass nucleus runtime 14
Greengrass nucleus 15
Greengrass nucleus lite 15
What's new in Version 2 18
AWS IoT Greengrass Core v2.18.2 software update 20
Public component updates 21
AWS IoT Greengrass Core v2.18.1 software update 22
Public component updates 22
AWS IoT Greengrass Core v2.18.0 software update 23
Public component updates 23
Greengrass nucleus lite v2.5.1 software update 26
Public component updates 26
AWS IoT Greengrass Core v2.17.0 software update 27
Public component updates 27
AWS IoT Greengrass Core v2.16.1 software update 29
Public component updates 29
AWS IoT Greengrass Core v2.16.0 software update 30
Public component updates 30
AWS IoT Greengrass Core v2.15.1 software update 32
Public component updates 32
AWS IoT Greengrass Core v2.15.0 software update 33
Public component updates 33
AWS IoT Greengrass Core v2.14.3 software update 35
Public component updates 36
AWS IoT Greengrass Core v2.14.2 software update 37
Public component updates 37

AWS IoT Greengrass Core v2.14.1 software update 38
Public component updates 39
AWS IoT Greengrass Core v2.14.0 software update 40
Public component updates 41
AWS IoT Greengrass Core v2.13.0 software update 44
Public component updates 44
AWS IoT Greengrass Core v2.12.6 software update 46
Public component updates 46
AWS IoT Greengrass Core v2.12.5 software update 47
Public component updates 47
AWS IoT Greengrass Core v2.12.4 software update 48
Public component updates 49
AWS IoT Greengrass Core v2.12.3 software update 49
Public component updates 50
AWS IoT Greengrass Core v2.12.2 software update 51
Public component updates 52
AWS IoT Greengrass Core v2.12.1 software update 53
Public component updates 53
AWS IoT Greengrass Core v2.12.0 software update 55
Public component updates 55
AWS IoT Greengrass Core v2.11.3 software update 56
Public component updates 56
AWS IoT Greengrass Core v2.11.2 software update 58
Public component updates 58
AWS IoT Greengrass Core v2.11.1 software update 59
Public component updates 59
AWS IoT Greengrass Core v2.11.0 software update 60
Public component updates 60
AWS IoT Greengrass Core v2.10.3 software update 62
Public component updates 62
AWS IoT Greengrass Core v2.10.2 software update 63
Public component updates 63
AWS IoT Greengrass Core v2.10.1 software update 65
Public component updates 65
AWS IoT Greengrass Core v2.10.0 software update 66
Public component updates 67

AWS IoT Greengrass Core v2.9.6 software update 68
Public component updates 68
AWS IoT Greengrass Core v2.9.5 software update 69
Public component updates 69
AWS IoT Greengrass Core v2.9.4 software update 70
Public component updates 71
AWS IoT Greengrass Core v2.9.3 software update 72
Public component updates 72
AWS IoT Greengrass Core v2.9.2 software update 73
Public component updates 73
AWS IoT Greengrass Core v2.9.1 software update 74
Public component updates 74
AWS IoT Greengrass Core v2.9.0 software update 76
Public component updates 76
AWS IoT Greengrass Core v2.8.1 software update 78
Public component updates 78
AWS IoT Greengrass Core v2.8.0 software update 79
Public component updates 80
AWS IoT Greengrass Core v2.7.0 software update 82
Public component updates 82
AWS IoT Greengrass Core v2.6.0 software update 84
Public component updates 85
AWS IoT Greengrass Core v2.5.6 software update 89
Public component updates 89
AWS IoT Greengrass Core v2.5.5 software update 90
Public component updates 90
AWS IoT Greengrass Core v2.5.4 software update 91
Public component updates 92
AWS IoT Greengrass Core v2.5.3 software update 93
Public component updates 93
AWS IoT Greengrass Core v2.5.2 software update 94
Public component updates 94
AWS IoT Greengrass Core v2.5.1 software update 96
Public component updates 96
AWS IoT Greengrass Core v2.5.0 software update 97
Platform support updates 98

Public component updates ... 98
AWS IoT Greengrass Core v2.4.0 software update ... 102
Public component updates ... 103
AWS IoT Greengrass Core v2.3.0 software update ... 105
Public component updates ... 106
AWS IoT Greengrass Core v2.2.0 software update ... 107
Public component updates ... 107
AWS IoT Greengrass Core v2.1.0 software update ... 110
Platform support updates ... 111
Public component updates ... 111
AWS IoT Greengrass Core v2.0.5 software update ... 118
Public component updates ... 118
AWS IoT Greengrass Core v2.0.4 software update ... 119
Public component updates ... 120
Migrate from Version 1 ... 122
Migration overview ... 122
Differences between V1 and V2 ... 124
Choose your runtime ... 134
Event source compatibility matrix ... 134
Runtime selection decision flow ... 135
Next steps ... 135
Set up test device ... 136
Set up test device (Greengrass nucleus) ... 136
Set up test device (Greengrass nucleus lite) ... 141
Upgrade V1 core devices to V2 ... 210
Step 1: Install the AWS IoT Greengrass Core software v2.x ... 211
Step 2: Deploy Greengrass V2 components to the core devices ... 214
Getting started ... 217
Prerequisites ... 218
Sign up for an AWS account ... 219
Step 2: Set up your environment ... 220
Step 3: Install the AWS IoT Greengrass Core software ... 226
Install the AWS IoT Greengrass Core software (console) ... 226
Install the AWS IoT Greengrass Core software (CLI) ... 233
Run the Greengrass software (Linux) ... 238
Verify the Greengrass CLI installation on the device ... 239

Step 4: Develop and test a component on your device ... 240
Step 5: Create your component in the AWS IoT Greengrass service ... 252
Step 6: Deploy your component ... 264
Next steps ... 269
Setting up Greengrass core devices ... 270
Supported platforms ... 270
Device requirements ... 270
Lambda function requirements ... 271
Set up an AWS account ... 273
Install the AWS IoT Greengrass Core software ... 273
Install with automatic provisioning ... 275
Install with manual provisioning ... 291
Install with fleet provisioning ... 329
Install with custom provisioning ... 375
Installer arguments ... 392
Run the AWS IoT Greengrass Core software ... 397
Check if the AWS IoT Greengrass Core software runs as a system service ... 397
Run the AWS IoT Greengrass Core software as a system service ... 399
Run the AWS IoT Greengrass Core software without a system service ... 399
Run AWS IoT Greengrass in Docker ... 400
Supported platforms and requirements ... 401
Software downloads ... 401
Choose how to provision AWS resources ... 402
Build the AWS IoT Greengrass image from a Dockerfile ... 402
Run AWS IoT Greengrass in Docker with automatic provisioning ... 408
Run AWS IoT Greengrass in Docker with manual provisioning ... 416
Troubleshooting AWS IoT Greengrass in a Docker container ... 435
Configure the AWS IoT Greengrass Core software ... 438
Deploy the Greengrass nucleus component ... 439
Configure the Greengrass nucleus as a system service ... 439
Control memory allocation with JVM options ... 443
Configure the user that runs components ... 445
Configure system resource limits ... 449
Connect on port 443 or through a network proxy ... 451
Use a device certificate signed by a private CA ... 459
Configure MQTT timeouts and cache settings ... 459

Configure Greengrass Nucleus on IPv6 network ... 459
Update the AWS IoT Greengrass Core software (OTA) ... 460
Requirements ... 461
Considerations for core devices ... 461
Greengrass nucleus update behavior ... 462
Perform an OTA update ... 464
Uninstall the AWS IoT Greengrass Core software ... 464
Tutorials ... 468
Video tutorials ... 468
Videos ... 468
Develop a component that defers component updates ... 469
Prerequisites ... 470
Step 1: Install the Greengrass Development Kit CLI ... 471
Step 2: Develop a component that defers updates ... 472
Step 3: Publish the component to the AWS IoT Greengrass service ... 481
Step 4: Deploy and test the component on a core device ... 484
Interact with local IoT devices over MQTT ... 489
Prerequisites ... 490
Step 1: Review and update the core device AWS IoT policy ... 491
Step 2: Enable client device support ... 492
Step 3: Connect client devices ... 498
Step 4: Develop a component that communicates with client devices ... 501
Step 5: Develop a component that interacts with client device shadows ... 508
Secure Greengrass Nucleus with TPM ... 534
Prerequisites ... 535
Step 1: Install TPM2 tools and dependencies ... 535
Step 2: Initialize PKCS#11 store and create a slot ... 536
Step 3: Create a token and a key ... 537
Step 4: Generate Certificate Signing Request(CSR) ... 539
Step 5: Create the thing certificate ... 540
Step 6: Import thing certificate to TPM ... 540
Step 7: Capture the PKCS#11 object URL ... 541
Step 8: Configure and install Greengrass with TPM2 support ... 542
Step 9: Verify the installation ... 543
Troubleshooting ... 544
Next steps ... 545

Secure Greengrass Nucleus Lite with TPM ... 545
Prerequisites ... 545
Step 1: Set up a NitroTPM instance ... 546
Step 2: Install and configure TPM tools ... 546
Step 3: Configure the OpenSSL TPM2 provider ... 546
Step 4: Generate persistent TPM keys ... 547
Step 5: Generate Certificate Signing Request (CSR) ... 548
Step 6: Create the certificate from CSR ... 548
Step 7: Set up Greengrass Nucleus Lite with TPM support ... 549
Troubleshooting ... 550
Next steps ... 551
Get started with SageMaker AI Edge Manager ... 551
Prerequisites ... 552
Set up in SageMaker AI Edge Manager ... 554
Create the sample components ... 555
Run sample image classification inference ... 556
Perform sample image classification inference ... 560
Prerequisites ... 561
Step 1: Subscribe to the default notifications topic ... 562
Step 2: Deploy the TensorFlow Lite image classification component ... 563
Step 3: View inference results ... 564
Next steps ... 566
Perform sample image classification inference on images from a camera ... 567
Prerequisites ... 568
Step 1: Configure the camera module on your device ... 569
Step 2: Verify your subscription to the default notifications topic ... 571
Step 3: Modify the TensorFlow Lite image classification component configuration and deploy it ... 571
Step 4: View inference results ... 573
Next steps ... 574
Set up high availability with Pacemaker ... 574
Prerequisites and cluster setup ... 576
Active/passive AWS IoT Greengrass V2 service ... 584
Active/passive load balancer ... 589
Active/active AWS IoT Greengrass V2 component ... 593
Switch a Greengrass core device to a different AWS account or Region ... 597

Prerequisites ... 598
Create the deployment ... 599
What to expect ... 600
Verify the switch ... 601
Components ... 602
AWS-provided components ... 602
Greengrass nucleus ... 616
Greengrass nucleus lite ... 661
Client device auth ... 668
CloudWatch metrics ... 743
AWS IoT Device Defender ... 772
Disk spooler ... 789
Docker application manager ... 794
Edge connector for Kinesis Video Streams ... 804
Greengrass CLI ... 812
IP detector ... 827
Firehose ... 838
Lambda launcher ... 856
Lambda manager ... 859
Lambda runtimes ... 869
Legacy subscription router ... 871
Local debug console ... 883
Log manager ... 900
Machine learning components ... 949
Modbus-RTU protocol adapter ... 1063
MQTT bridge ... 1095
MQTT 3.1.1 broker (Moquette) ... 1119
MQTT 5 broker (EMQX) ... 1126
Nucleus telemetry emitter ... 1143
PKCS#11 provider ... 1156
Secret manager ... 1165
Secure tunneling ... 1178
Shadow manager ... 1193
Amazon SNS ... 1223
Stream manager ... 1241
System log forwarder ... 1255

Systems Manager Agent 1261
Token exchange service 1268
IoT SiteWise OPC UA collector 1272
IoT SiteWise OPC UA data source simulator 1282
IoT SiteWise publisher 1285
IoT SiteWise processor 1299
Publisher-supported components 1315
AIShield.Edge 1315
AI EdgeLabs Sensor 1316
Greengrass S3 Ingestor 1316
Community components 1317
Greengrass development tools 1321
Greengrass Development Kit CLI 1322
Greengrass Command Line Interface 1352
Use Greengrass Testing Framework 1370
Develop components 1386
Component lifecycle 1387
Component types 1388
Create components 1389
Test components with local deployments 1402
Publish components to deploy 1404
Interact with AWS services 1411
Run a Docker container 1414
Recipe reference 1437
Environment variables 1469
Deploy components to devices 1471
Core device deployments 1471
Platform dependency resolution 1471
Component dependency resolution 1471
Removing a device from a thing group 1479
Deployments 1480
Deployment options 1480
Create deployments 1482
Create subdeployments 1502
Revise deployments 1506
Cancel deployments 1508

Check deployment status ... 1509
Logging and monitoring ... 1513
Monitoring tools ... 1513
Monitor Greengrass logs ... 1514
Access file system logs ... 1515
Access CloudWatch Logs ... 1517
Access system service logs ... 1519
Enable logging to CloudWatch Logs ... 1521
Configure logging for AWS IoT Greengrass ... 1522
AWS CloudTrail logs ... 1524
Log API calls with CloudTrail ... 1524
AWS IoT Greengrass V2 information in CloudTrail ... 1525
AWS IoT Greengrass data events in CloudTrail ... 1526
AWS IoT Greengrass management events in CloudTrail ... 1530
Understanding AWS IoT Greengrass V2 log file entries ... 1530
Gather system health telemetry data ... 1532
Telemetry metrics ... 1533
Configure telemetry agent settings ... 1538
Subscribe to telemetry data in EventBridge ... 1538
Get deployment and component health status notifications ... 1547
Deployment status change event ... 1547
Component status change event ... 1549
Prerequisites for creating EventBridge rules ... 1551
Configure device health notifications (console) ... 1552
Configure device health notifications (CLI) ... 1553
Configure device health notifications (CloudFormation) ... 1554
See also ... 1554
Check core device status ... 1554
Check health of a core device ... 1555
Check health of a core device group ... 1556
Check core device component status ... 1556
Run Lambda functions ... 1558
Requirements ... 1559
Configure Lambda function lifecycle ... 1559
Configure Lambda function containerization ... 1560
Import a Lambda function as a component (console) ... 1563

Step 1: Choose a Lambda function to import 1563
Step 2: Configure Lambda function parameters 1564
Step 3: (Optional) Specify supported platforms for the Lambda function 1566
Step 4: (Optional) Specify component dependencies for the Lambda function 1567
Step 5: (Optional) Run the Lambda function in a container 1568
Step 6: Create the Lambda function component 1570
Import a Lambda function as a component (CLI) 1570
Step 1: Define the Lambda function configuration 1570
Step 2: Create the Lambda function component 1590
Communicate with the Greengrass nucleus, other components, and AWS IoT Core 1592
IPC client versions 1593
Supported SDKs 1594
Connect to the AWS IoT Greengrass Core IPC service 1594
Authorize components to perform IPC operations 1601
Wildcards in authorization policies 1602
Recipe variables in authorization policies 1603
Special characters in authorization policies 1603
Authorization policy examples 1604
Subscribe to IPC event streams 1607
Define subscription handlers 1608
Example subscription handlers 1611
IPC best practices 1623
Publish/subscribe local messages 1624
Minimum SDK versions 1624
Authorization 1625
PublishToTopic 1628
SubscribeToTopic 1638
Examples 1654
Publish/subscribe AWS IoT Core MQTT messages 1689
Minimum SDK versions 1689
Authorization 1690
PublishToIoTCore 1694
SubscribeToIoTCore 1706
SubscribeToIoTCoreConnectionStatus 1723
Examples 1725
Interact with component lifecycle 1745

Minimum SDK versions 1745
Authorization 1746
UpdateState 1747
SubscribeToComponentUpdates 1749
DeferComponentUpdate 1751
PauseComponent 1752
ResumeComponent 1754
Interact with component configuration 1755
Minimum SDK versions 1756
GetConfiguration 1756
UpdateConfiguration 1760
SubscribeToConfigurationUpdate 1763
SubscribeToValidateConfigurationUpdates 1768
SendConfigurationValidityReport 1769
Retrieve secret values 1770
Minimum SDK versions 1770
Authorization 1771
GetSecretValue 1772
Examples 1778
Interact with local shadows 1784
Minimum SDK versions 1785
Authorization 1786
GetThingShadow 1797
UpdateThingShadow 1804
DeleteThingShadow 1813
ListNamedShadowsForThing 1818
Manage local deployments and components 1826
Minimum SDK versions 1827
Authorization 1827
CreateLocalDeployment 1830
ListLocalDeployments 1833
GetLocalDeploymentStatus 1834
ListComponents 1835
GetComponentDetails 1836
RestartComponent 1837
StopComponent 1840

CreateDebugPassword ... 1841
Authenticate and authorize client devices ... 1842
Minimum SDK versions ... 1843
Authorization ... 1843
VerifyClientDeviceIdentity ... 1845
GetClientDeviceAuthToken ... 1845
AuthorizeClientDeviceAction ... 1847
SubscribeToCertificateUpdates ... 1847
Interact with local IoT devices ... 1850
Client device components ... 1850
Connect client devices to core devices ... 1853
Requirements ... 1854
Greengrass components for client device support ... 1867
Configure cloud discovery (console) ... 1869
Configure cloud discovery (AWS CLI) ... 1869
Associate client devices ... 1869
Authenticating clients while offline ... 1872
Manage core device endpoints ... 1873
Choose an MQTT broker ... 1880
Connecting to an MQTT broker ... 1881
Test communications ... 1883
Greengrass discovery RESTful API ... 1895
Relay MQTT messages between client devices and AWS IoT Core ... 1901
Configure and deploy the MQTT bridge component ... 1902
Relay MQTT messages ... 1903
Interact with client devices in components ... 1904
Configure and deploy the MQTT bridge component ... 1905
Receive MQTT messages from client devices ... 1906
Send MQTT messages to client devices ... 1907
Interact with and sync client device shadows ... 1907
Prerequisites ... 1908
Enable shadow manager to communicate with client devices ... 1908
Interact with client device shadows in components ... 1911
Sync client device shadows with AWS IoT Core ... 1911
Use IPv6 for local messaging ... 1911
Configure IP detector to use IPv6 ... 1912

Troubleshooting 1916
Greengrass discovery issues 1916
MQTT connection issues 1924
Interact with device shadows 1930
Interact with shadows in components 1930
Retrieve and modify shadow states 1931
React to shadow state changes 1932
Sync local device shadows with AWS IoT Core 1933
Prerequisites 1933
Configure the shadow manager component 1934
Sync local shadows 1935
Shadow merge conflict behavior 1936
Manage data streams 1937
Stream management workflow 1938
Requirements 1939
Data security 1939
Local data security 1940
Client authentication 1940
See also 1941
Configure stream manager 1941
Stream manager parameters 1941
See also 1944
Create custom components that use stream manager 1944
Define component recipes that use stream manager 1945
Connect to stream manager in application code 1957
Use StreamManagerClient to work with streams 1960
Create message stream 1961
Append message 1965
Read messages 1971
List streams 1974
Describe message stream 1975
Update message stream 1977
Delete message stream 1981
See also 1983
Export configurations for supported cloud destinations 1983
Perform machine learning inference 1999

How AWS IoT Greengrass ML inference works ... 1999
What's different in AWS IoT Greengrass Version 2? ... 2001
Requirements ... 2001
Supported model sources ... 2001
Supported runtimes ... 2002
Machine learning components ... 2002
Use SageMaker AI Edge Manager ... 2009
How it works ... 2009
Requirements ... 2010
Get started with SageMaker AI Edge Manager ... 2012
Customize your machine learning components ... 2012
Modify the configuration of a public inference component ... 2013
Use a custom model with the sample inference component ... 2015
Create custom machine learning components ... 2019
Create a custom inference component ... 2021
Troubleshooting ... 2028
Failed to fetch library ... 2030
Cannot open shared object file ... 2030
Error: ModuleNotFoundError: No module named '<library>' ... 2030
No CUDA-capable device is detected ... 2032
No such file or directory ... 2032
RuntimeError: module compiled against API version 0xf but this version of NumPy is <version> ... 2033
picamera.exc.PiCameraError: Camera is not enabled ... 2033
Memory errors ... 2034
Disk space errors ... 2034
Timeout errors ... 2034
Manage core devices with AWS Systems Manager ... 2035
Install the Systems Manager Agent ... 2036
Step 1: Complete general Systems Manager setup steps ... 2036
Step 2: Create an IAM service role for Systems Manager ... 2036
Step 3: Add permissions to the token exchange role ... 2037
Step 4: Deploy the Systems Manager Agent component ... 2041
Step 5: Verify core device registration with Systems Manager ... 2044
Uninstall the Systems Manager Agent ... 2045
Step 1: Deregister the core device from Systems Manager ... 2046

Step 2: Uninstall the Systems Manager Agent component ... 2046
Step 3: Uninstall the Systems Manager Agent software ... 2047
Security ... 2048
Data protection ... 2049
Data encryption ... 2050
Hardware security integration ... 2052
Device authentication and authorization ... 2064
X.509 certificates ... 2065
AWS IoT policies ... 2066
Update a core device's AWS IoT policy ... 2071
Minimal AWS IoT policy ... 2076
Minimal AWS IoT policy to support client devices ... 2079
Minimal AWS IoT policy for client devices ... 2081
Identity and access management ... 2083
Audience ... 2083
Authenticating with identities ... 2083
Managing access using policies ... 2084
See also ... 2086
How AWS IoT Greengrass works with IAM ... 2086
Identity-based policy examples ... 2091
Authorize core devices to interact with AWS services ... 2093
Minimal IAM policy for installer to provision resources ... 2098
Greengrass service role ... 2102
AWS managed policies ... 2111
Cross-service confused deputy prevention ... 2117
Troubleshooting identity and access issues ... 2118
Allow device traffic through a proxy or firewall ... 2120
Endpoints for basic operation ... 2120
Endpoints for installation with automatic provisioning ... 2125
Endpoints for AWS-provided components ... 2126
Compliance validation ... 2126
FIPS endpoints ... 2127
Enable FIPS endpoints with deployment ... 2128
Install Nucleus with FIPS endpoints with manual resource provisioning ... 2130
Install FIPS endpoints with fleet provisioning ... 2162
Install FIPS endpoints with auto resource provisioning ... 2180

FIPS compliance first party components ... 2196
Resilience ... 2198
Infrastructure security ... 2198
Configuration and vulnerability analysis ... 2199
Code integrity ... 2200
VPC endpoints (AWS PrivateLink) ... 2201
Considerations for AWS IoT Greengrass VPC endpoints ... 2202
Create an interface VPC endpoint for AWS IoT Greengrass control plane operations ... 2202
Creating a VPC endpoint policy for AWS IoT Greengrass ... 2202
Operate an AWS IoT Greengrass core device in VPC ... 2203
Security best practices ... 2208
Grant minimum possible permissions ... 2208
Don't hardcode credentials in Greengrass components ... 2209
Don't log sensitive information ... 2209
Keep your device clock in sync ... 2210
Cipher Suite Recommendations ... 2210
See also ... 2210
Run AWS IoT Greengrass V2 as non-root ... 2210
Set up AWS IoT Greengrass V2 core devices as non-root ... 2211
Migrate AWS IoT Greengrass V2 core devices to non-root ... 2226
Using AWS IoT Device Tester for AWS IoT Greengrass V2 ... 2233
AWS IoT Greengrass qualification suite ... 2233
Custom test suites ... 2234
Supported versions ... 2234
Latest IDT version for AWS IoT Greengrass V2 ... 2235
Earlier IDT versions for AWS IoT Greengrass ... 2235
Unsupported versions of AWS IoT Device Tester for AWS IoT Greengrass V2 ... 2236
Download IDT for AWS IoT Greengrass V2 ... 2241
Download IDT manually ... 2242
Download IDT programmatically ... 2243
Use IDT to run the AWS IoT Greengrass qualification suite ... 2248
Test suite versions ... 2249
Test group descriptions ... 2249
Prerequisites ... 2252
Configure your device to run IDT tests ... 2272
Configure IDT settings ... 2282

Run the AWS IoT Greengrass qualification suite ... 2299
Understanding results and logs ... 2302
Use IDT to develop and run your own test suites ... 2306
Download the latest version of IDT for AWS IoT Greengrass ... 2252
Test suite creation workflow ... 2307
Tutorial: Build and run the sample IDT test suite ... 2308
Tutorial: Develop a simple IDT test suite ... 2313
Create IDT test suite configuration files ... 2322
Configure the IDT test orchestrator ... 2330
Configure the IDT state machine ... 2337
Create IDT test case executables ... 2361
Use the IDT context ... 2368
Configure settings for test runners ... 2372
Debug and run custom test suites ... 2384
Review IDT test results and logs ... 2386
IDT usage metrics ... 2393
Troubleshooting IDT for AWS IoT Greengrass V2 ... 2398
Where to look for errors ... 2398
Resolving IDT for AWS IoT Greengrass V2 errors ... 2399
Support policy for AWS IoT Device Tester for AWS IoT Greengrass ... 2406
Greengrass based IoT solutions ... 2408
Eurotech ... 2408
Troubleshooting ... 2409
View AWS IoT Greengrass Core software and component logs ... 2409
AWS IoT Greengrass Core software issues ... 2409
ThrottlingException from ListDeployments API ... 2411
Unable to set up core device ... 2411
Unable to start the AWS IoT Greengrass Core software as a system service ... 2411
Unable to set up nucleus as a system service ... 2411
Unable to connect to AWS IoT Core ... 2412
Out of memory error ... 2412
Unable to install Greengrass CLI ... 2412
User root is not allowed to execute ... 2413
com.aws.greengrass.lifecyclemanager.GenericExternalService: Could not determine user/group to run with ... 2413
Failed to map segment from shared object: operation not permitted ... 2413

Failed to set up Windows service 2414
com.aws.greengrass.util.exceptions.TLSAuthException: Failed to get trust manager 2414
com.aws.greengrass.deployment.IotJobsHelper: No connection available during subscribing to Iot Jobs descriptions topic. Will retry in sometime 2415
software.amazon.awssdk.services.iam.model.IamException: The security token included in the request is invalid 2415
software.amazon.awssdk.services.iot.model.IotException: User: <user> is not authorized to perform: iot:GetPolicy 2416
Error: com.aws.greengrass.shadowmanager.sync.model.FullShadowSyncRequest: Could not execute cloud shadow get request 2416
Operation aws.greengrass#<operation> is not supported by Greengrass 2417
java.io.FileNotFoundException: <stream-manager-store-root-dir>/stream_manager_metadata_store (Permission denied) 2418
com.aws.greengrass.security.provider.pkcs11.PKCS11CryptoKeyService: Private key or certificate with label <label> does not exist 2418
software.amazon.awssdk.services.secretsmanager.model.SecretsManagerException: User: <user> is not authorized to perform: secretsmanager:GetSecretValue on resource: <arn> 2418
software.amazon.awssdk.services.secretsmanager.model.SecretsManagerException: Access to KMS is not allowed 2419
java.lang.NoClassDefFoundError: com/aws/greengrass/security/CryptoKeySpi 2420
com.aws.greengrass.security.provider.pkcs11.PKCS11CryptoKeyService: CKR_OPERATION_NOT_INITIALIZED 2420
Greengrass core device stuck on nucleus v2.12.3 2420
Greengrass nucleus v2.14.0 systemd template issue 2422
AWS IoT Greengrass cloud issues 2423
An error occurred (AccessDeniedException) when calling the CreateComponentVersion operation: User: arn:aws:iam::123456789012:user/<username> is not authorized to perform: null 2423
Invalid Input: Encountered following errors in Artifacts: {<s3ArtifactUri> = Specified artifact resource cannot be accessed} 2423
INACTIVE deployment status 2424
CreateDeployment returns a success response but GetDeployment returns a 404 error ... 2424
Core device deployment issues 2425
Error: com.aws.greengrass.componentmanager.exceptions.PackageDownloadException: Failed to download artifact 2426

Error:
com.aws.greengrass.componentmanager.exceptions.ArtifactChecksumMismatchException:
Integrity check for downloaded artifact failed. Probably due to file corruption. 2427
Error:
com.aws.greengrass.componentmanager.exceptions.NoAvailableComponentVersionException:
Failed to negotiate component <name> version with cloud and no local applicable version
satisfying requirement <requirements> 2428
software.amazon.awssdk.services.greengrassv2data.model.ResourceNotFoundException:
The latest version of Component <componentName> doesn't claim platform
<coreDevicePlatform> compatibility 2429
com.aws.greengrass.componentmanager.exceptions.PackagingException: The deployment
attempts to update the nucleus from aws.greengrass.Nucleus-<version> to
aws.greengrass.Nucleus-<version> but no component of type nucleus was included as
target component 2429
Error: com.aws.greengrass.deployment.exceptions.DeploymentException: Unable to
process deployment. Greengrass launch directory is not set up or Greengrass is not set up
as a system service 2430
Info:
com.aws.greengrass.deployment.exceptions.RetryableDeploymentDocumentDownloadException:
Greengrass Cloud Service returned an error when getting full deployment configuration 2431
Warn: com.aws.greengrass.deployment.DeploymentService: Failed to get thing group
hierarchy 2431
Deployment unexpectedly removes components from a core device 2432
Info: com.aws.greengrass.deployment.DeploymentDocumentDownloader: Calling
Greengrass cloud to get full deployment configuration 2435
Caused by:
software.amazon.awssdk.services.greengrassv2data.model.GreengrassV2DataException:
null (Service: GreengrassV2Data, Status Code: 403, Request ID: <some_request_id>,
Extended Request ID: null) 2435
Core device component issues 2435
Warn: '<command>' is not recognized as an internal or external command 2436
Python script doesn't log messages 2437
Component configuration doesn't update when changing default configuration 2438
awsiot.greengrasscoreipc.model.UnauthorizedError 2439
com.aws.greengrass.authorization.exceptions.AuthorizationException: Duplicate policy ID
"<id>" for principal "<componentList>" 2439

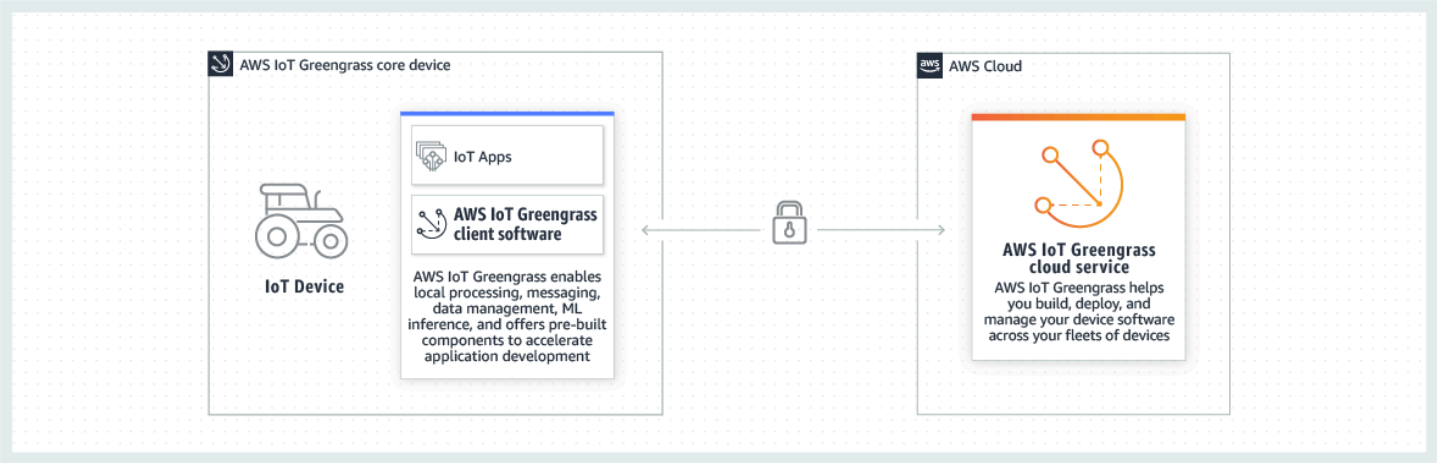
com.aws.greengrass.tes.CredentialRequestHandler: Error in retrieving AwsCredentials from TES (HTTP 400) ... 2440
com.aws.greengrass.tes.CredentialRequestHandler: Error in retrieving AwsCredentials from TES (HTTP 403) ... 2442
com.aws.greengrass.tes.CredentialsProviderError: Could not load credentials from any providers ... 2442
Received error when attempting to retrieve ECS metadata: Could not connect to the endpoint URL: "<tokenExchangeServiceEndpoint>" ... 2443
copyFrom: <configurationPath> is already a container, not a leaf ... 2443
com.aws.greengrass.componentmanager.plugins.docker.exceptions.DockerLoginException: Error logging into the registry using credentials - 'The stub received bad data.' ... 2443
java.io.IOException: Cannot run program "cmd" ...: [LogonUser] The password for this account has expired. ... 2444
aws.greengrass.StreamManager: Instant exceeds minimum or maximum instant ... 2445
Core device Lambda function component issues ... 2446
The following cgroup subsystems are not mounted: devices, memory ... 2446
ipc_client.py:64,HTTP Error 400:Bad Request, b'No subscription exists for the source <label-or-lambda-arn> and subject <label-or-lambda-arn> ... 2447
Component version discontinued ... 2447
Greengrass CLI issues ... 2448
java.lang.RuntimeException: Unable to create ipc client ... 2448
AWS CLI issues ... 2448
Error: Invalid choice: 'greengrassv2' ... 2448
Detailed deployment error codes ... 2449
Permission error ... 2450
Request error ... 2452
Component recipe error ... 2454
AWS component error, user component error, component error ... 2456
Device error ... 2457
Dependency error ... 2458
HTTP error ... 2459
Network error ... 2460
Nucleus error ... 2460
Server error ... 2462
Cloud service error ... 2462
Generic errors ... 2463

Unknown error ... 2464
Detailed component status codes ... 2465
Tag your resources ... 2468
Using tags in AWS IoT Greengrass V2 ... 2468
Tag with the AWS Management Console ... 2468
Tag with the AWS IoT Greengrass V2 API ... 2468
Using tags with IAM policies ... 2469
AWS CloudFormation resources ... 2471
AWS IoT Greengrass and CloudFormation templates ... 2471
ComponentVersion template example ... 2471
Deployment template example ... 2472
Learn more about CloudFormation ... 2473
Open source software ... 2474
Document history ... 2475
AWS Glossary ... 2536

What is AWS IoT Greengrass?

AWS IoT Greengrass is an open source Internet of Things (IoT) edge runtime and cloud service that helps you build, deploy and manage IoT applications on your devices. You can use AWS IoT Greengrass to build software that enables your devices to act locally on the data that they generate, run predictions based on machine learning models, and filter and aggregate device data. AWS IoT Greengrass enables your devices to collect and analyze data closer to where that data is generated, react autonomously to local events, and communicate securely with other devices on the local network. Greengrass devices can also communicate securely with AWS IoT Core and export IoT data to the AWS Cloud. You can use AWS IoT Greengrass to build edge applications using pre-built software modules, called components, that can connect your edge devices to AWS services or third-party services. You can also use AWS IoT Greengrass to package and run your software using Lambda functions, Docker containers, native operating system processes, or custom runtimes of your choice.

The following example shows how an AWS IoT Greengrass device interacts with the AWS Cloud.



New features

AWS IoT Greengrass V2 introduces new features and improvements. The following includes more information about the new features offered in version 2.

- What's new in AWS IoT Greengrass Version 2

For first-time users of AWS IoT Greengrass

If you're new to AWS IoT Greengrass, we recommend that you review the following section:

- How AWS IoT Greengrass works

Next, follow the getting started tutorial to try out the basic features of AWS IoT Greengrass. In this tutorial, you install the AWS IoT Greengrass Core software on a device, develop a Hello World component, and package that component for deployment.

For existing users of AWS IoT Greengrass V1

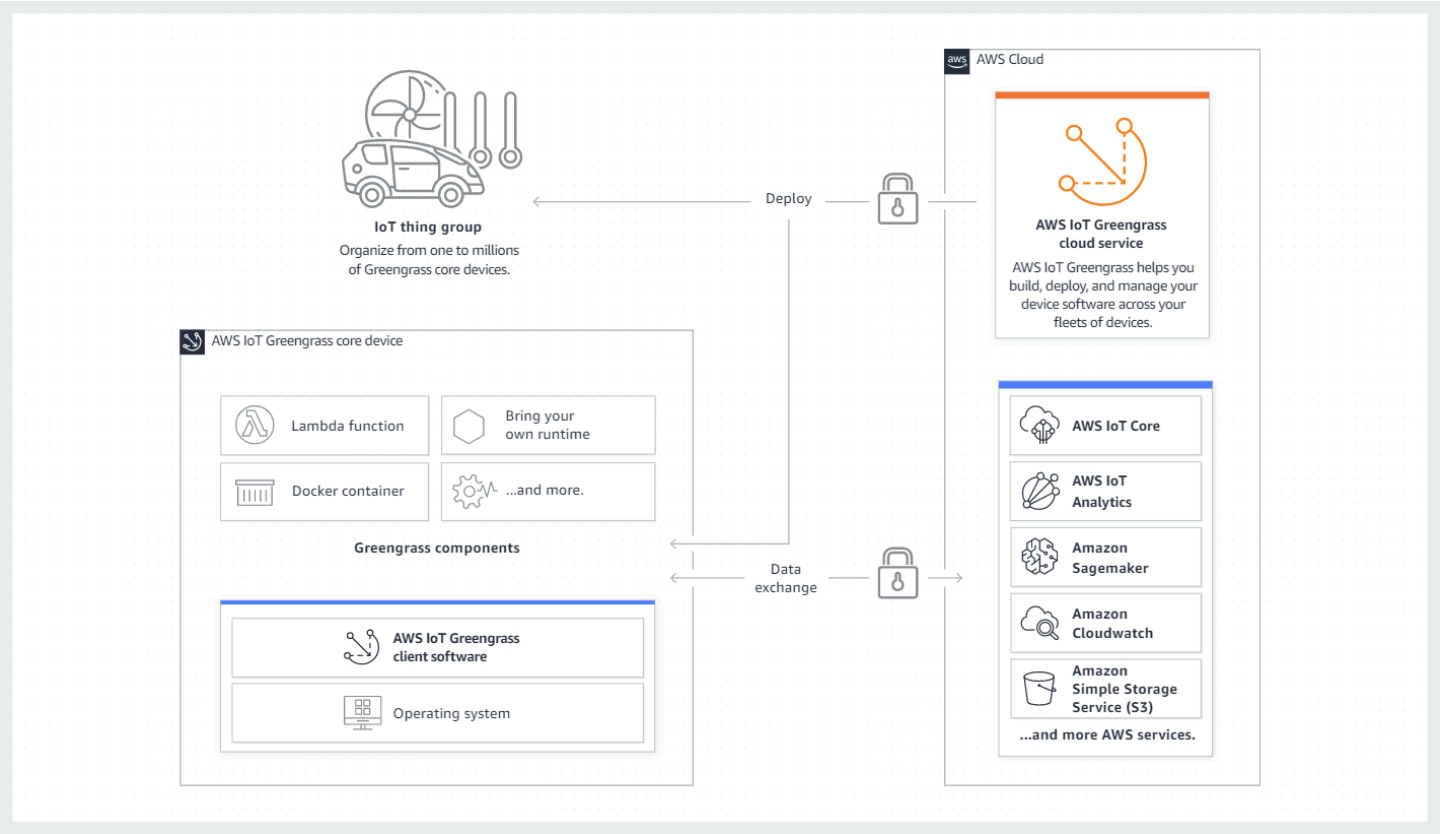
End of support notice: On October 7, 2026, AWS ended support for AWS IoT Greengrass Version 1. The AWS IoT Greengrass V1 API operations and resources are no longer available. For more information about how to move from version 1 to version 2, see Migrate from AWS IoT Greengrass Version 1.

How AWS IoT Greengrass works

The AWS IoT Greengrass client software, also called AWS IoT Greengrass Core software, runs on Windows and Linux-based distributions, such as Ubuntu or Raspberry Pi OS, for devices with ARM or x86 architectures. With AWS IoT Greengrass, you can program devices to act locally on the data they generate, run predictions based on machine learning models, and filter and aggregate device data. AWS IoT Greengrass enables local execution of AWS Lambda functions, Docker containers, native OS processes, or custom runtimes of your choice.

AWS IoT Greengrass provides pre-built software modules called components that let you easily extend edge device functionality. AWS IoT Greengrass components enable you to connect to AWS services and third-party applications at the edge. After you develop your IoT applications, AWS IoT Greengrass enables you to remotely deploy, configure, and manage those applications on your fleet of devices in the field.

The following example shows how an AWS IoT Greengrass device interacts with the AWS IoT Greengrass cloud service and other AWS services in the AWS Cloud.



Key concepts for AWS IoT Greengrass

The following are essential concepts for understanding and using AWS IoT Greengrass:

AWS IoT thing

An AWS IoT thing is a representation of a specific device or logical entity. Information about a thing is stored in the AWS IoT registry.

Greengrass core device

A device that runs the AWS IoT Greengrass Core software. A Greengrass core device is an AWS IoT thing. You can add multiple core devices to AWS IoT thing groups to create and manage groups of Greengrass core devices. For more information, see Setting up AWS IoT Greengrass core devices.

Greengrass client device

A device that connects to and communicates with a Greengrass core device over MQTT. A Greengrass client device is an AWS IoT thing. The core device can process, filter, and aggregate data from client devices that connect to it. You can configure the core device to relay MQTT

messages between client devices, the AWS IoT Core cloud service, and Greengrass components. For more information, see Interact with local IoT devices.

Client devices can run FreeRTOS or use the AWS IoT Device SDK or Greengrass discovery API to get information about core devices to which they can connect.

Greengrass component

A software module that is deployed to and runs on a Greengrass core device. All software that is developed and deployed with AWS IoT Greengrass is modeled as a component. AWS IoT Greengrass provides pre-built public components that provide features and functionality that you can use in your applications. You can also develop your own custom components, on your local device or in the cloud. After you develop a custom component, you can use the AWS IoT Greengrass cloud service to deploy it to single or multiple core devices. You can create a custom component and deploy that component to a core device. When you do, the core device downloads the following resources to run the component:

- **Recipe**: A JSON or YAML file that describes the software module by defining component details, configuration, and parameters.
- **Artifact**: The source code, binaries, or scripts that define the software that will run on your device. You can create artifacts from scratch, or you can create a component using a Lambda function, a Docker container, or a custom runtime.
- **Dependency**: The relationship between components that enables you to enforce automatic updates or restarts of dependent components. For example, you can have a secure message processing component dependent on an encryption component. This ensures that any updates to the encryption component automatically update and restart the message processing component.

For more information, see AWS-provided components and Develop AWS IoT Greengrass components.

Deployment

The process to send components and apply the desired component configuration to a destination target device, which can be a single Greengrass core device or a group of Greengrass core devices. Deployments automatically apply any updated component configurations to the target and include any other components that are defined as dependencies. You can also clone an existing deployment to create a new deployment that uses the same components but is deployed to a different target. Deployments are continuous, which means that any updates you make to the components or the component configuration of a deployment automatically

get sent to all destination targets. For more information, see Deploy AWS IoT Greengrass components to devices.

AWS IoT Greengrass Core software

As of version 2.14, AWS IoT Greengrass provides two alternative implementations of its device runtime, an executable known as the nucleus. The first, and previously only, nucleus is implemented in Java. This choice provides the greatest portability across architectures and operating systems. However, it also comes with a dependency on the Java Virtual Machine, resulting in a large memory footprint.

The second, newly added nucleus is implemented in C. This choice considerably reduces its footprint. However, it requires distribution (or compilation from source) separately for different target architectures and operating systems. When there is a need to distinguish the two, we will refer to the first implementation as the *Greengrass nucleus* and the latter as the *Greengrass nucleus lite*.

- **Optional components**: These configurable components are provided by AWS IoT Greengrass and enable additional features on your edge devices. Depending on your requirements, you can choose the optional components that you want to deploy to your device, such as data streaming, local machine learning inference, or a local command line interface. For more information, see AWS-provided components.

You can upgrade your AWS IoT Greengrass Core software by deploying new versions of your components to your device.

Features of AWS IoT Greengrass

AWS IoT Greengrass Version 2 consists of the following elements:

- **Software distributions**
  - The Greengrass nucleus component, which is the minimum installation of the AWS IoT Greengrass Core software. This component manages deployments, orchestration, and lifecycle management of Greengrass components.
  - Additional optional AWS-provided components that integrate with services, protocols, and software.
  - Greengrass development tools, which you can use to create, test, build, publish, and deploy custom Greengrass components.

  - The AWS IoT Device SDK and the AWS IoT Greengrass Component SDK, which contain the interprocess communication (IPC) library for custom Greengrass components. The AWS IoT Device SDK also includes the Greengrass discovery library for client devices.
  - The Stream Manager SDK, which you can use to manage data streams on core devices.
- **Cloud service**
  - AWS IoT Greengrass V2 API
  - AWS IoT Greengrass V2 console

AWS IoT Greengrass Core software

You can use the AWS IoT Greengrass Core software that runs on your edge devices to do the following:

- Process data streams on the local device with automatic exports to the AWS Cloud. For more information, see Manage data streams on Greengrass core devices.
- Support MQTT messaging between AWS IoT and components. For more information, see Publish/subscribe AWS IoT Core MQTT messages.
- Interact with local devices that connect and communicate over MQTT. For more information, see Interact with local IoT devices.
- Support local publish and subscribe messaging between components. For more information, see Publish/subscribe local messages.
- Deploy and invoke components and Lambda functions. For more information, see Deploy AWS IoT Greengrass components to devices.
- Manage component lifecycles, such as with support for install and run scripts. For more information, see AWS IoT Greengrass component recipe reference.
- Perform secure, over-the-air (OTA) software updates of the AWS IoT Greengrass Core software and custom components. For more information, see Update the AWS IoT Greengrass Core software (OTA) and Deploy AWS IoT Greengrass components to devices.
- Provide secure, encrypted storage of local secrets and controlled access by components. For more information, see Secret manager.
- Secure connections between devices and the AWS Cloud with device authentication and authorization. For more information, see Device authentication and authorization for AWS IoT Greengrass.

You configure and manage Greengrass core devices through AWS IoT Greengrass APIs where you create continuous software deployments. For more information, see Deploy AWS IoT Greengrass components to devices.

Some features are supported on only certain platforms. For more information, see Greengrass feature compatibility.

For more information about supported platforms, requirements, and downloads, see Setting up AWS IoT Greengrass core devices.

By downloading this software, you agree to the Greengrass Core Software License Agreement.

Greengrass feature compatibility

AWS IoT Greengrass supports devices that run various operating systems. Some features are supported on only certain operating systems. Use the following tables to learn which features are available for each supported operating system. For more information about supported operating systems, requirements, and how to set up Greengrass core devices, see Setting up AWS IoT Greengrass core devices.

Messaging

Feature	Linux	Windows	Greengrass Nucleus Lite (Linux)
Exchange MQTT messages between AWS IoT and components	✓ Yes	✓ Yes	✓ Yes (except for MQTT5 extensions)
Exchange local publish/subscribe messages between components	✓ Yes	✓ Yes	✓ Yes
Interact with local IoT devices over MQTT	✓ Yes	✓ Yes	✗ No
Interact with local Modbus-RTU devices	✓ Yes	✗ No	✗ No

Feature	Linux	Windows	Greengrass Nucleus Lite (Linux)
using the Modbus-RTU component			

Security

Feature	Linux	Windows	Greengrass lite (Linux)
Secure connections with device authentication and authorization	✓ Yes	✓ Yes	✓ Yes
Deploy and access secure, encrypted secrets from AWS Secrets Manager	✓ Yes	✓ Yes	✗ No
Use a hardware security module (HSM) to securely store the device's private key and certificate	✓ Yes	✗ No	Yes
Audit core devices with AWS IoT Device Defender	✓ Yes	✓ Yes	✗ No
Use AWS credentials to interact with AWS services	✓ Yes	✓ Yes	✓ Yes

Installation

Feature	Linux	Windows	Greengrass lite (Linux)
Install AWS IoT Greengrass with automatic provisioning	✓ Yes	✓ Yes	✗ No
Install AWS IoT Greengrass with manual provisioning	✓ Yes	✓ Yes	✓ Yes
Install AWS IoT Greengrass with AWS IoT fleet provisioning	✓ Yes	✓ Yes	✓ Yes
Install AWS IoT Greengrass with custom provisioning	Yes	Yes	✓ Yes
Run AWS IoT Greengrass in a Docker container using a prebuilt Docker image	✓ Yes	✗ No	✗ No

Note

AWS IoT Greengrass can be installed and run in a systemd-enabled docker container.

Remote maintenance and updates

Feature	Linux	Windows	Greengass lite (Linux)
Perform secure, over-the-air (OTA) software updates	✓ Yes	✓ Yes	✓ Yes
Manage core devices with AWS Systems Manager	✓ Yes	✗ No	✗ No
Connect to core devices with AWS IoT secure tunneling	✓ Yes	✗ No	✓ Yes

Machine learning

Feature	Linux	Windows	Greengrass lite (Linux)
Perform machine learning inference using Amazon SageMaker AI Edge Manager	✓ Yes	✓ Yes	✗ No
Perform machine learning inference using Amazon Lookout for Vision	✓ Yes	✗ No	✗ No
Perform machine learning inference using DLR	✓ Yes	✓ Yes	✗ No

Feature	Linux	Windows	Greengrass lite (Linux)
Perform machine learning inference using TensorFlow	✓ Yes	✓ Yes	✗ No

Component features

Feature	Linux	Windows	Greengrass lite (Linux)
Deploy and invoke Lambda functions	✓ Yes	✗ No	✗ No
Run Docker containers in components	✓ Yes	✓ Yes	✓ Yes
Process and export high-volume data streams using stream manager	✓ Yes	✓ Yes	✓ Yes
Manage component lifecycles with lifecycle scripts	✓ Yes	✓ Yes	✓ Yes
Interact with device shadows	✓ Yes	✓ Yes	✗ No
Upload logs to Amazon CloudWatch Logs	✓ Yes	✓ Yes	✓ Yes
Upload data to Amazon CloudWatch metrics using the	✓ Yes	✓ Yes	✓ Yes

Feature	Linux	Windows	Greengrass lite (Linux)
CloudWatch metrics component			
Publish messages to Amazon Simple Notification Service using the Amazon SNS component	✓ Yes	✗ No	✗ No
Publish data to Amazon Kinesis Data Streams delivery streams using stream manager	✓ Yes	✓ Yes	✓ Yes
Publish data to Amazon Data Firehose delivery streams using the Firehose component	✓ Yes	✗ No	✗ No
Gather and act on real-time system telemetry metrics	✓ Yes	✓ Yes	✗ No
Configure system resource limits for component processes	✓ Yes	✗ No	✗ No
Pause and resume component processes	✓ Yes	✗ No	✗ No
Integrate with AWS IoT SiteWise using the AWS IoT SiteWise components	✓ Yes	✓ Yes	✗ No

Feature	Linux	Windows	Greengrass lite (Linux)
Publish video streams to Amazon Kinesis Video Streams using the edge connector for Kinesis Video Streams component	✓ Yes	✗ No	✗ No

Component development

Feature	Linux	Windows	Greengrass lite (Linux)
Develop components locally on core devices	✓ Yes	✓ Yes	✓ Yes
Interact with a core device using the AWS IoT Greengrass CLI	✓ Yes	✓ Yes	✗ No
Interact with a core device using the local debug console	✓ Yes	✓ Yes	✗ No
Use the AWS IoT Greengrass Component SDK in custom components	✓ Yes	✓ Yes	✓ Yes
Use the AWS IoT Device SDK for Python in custom components	✓ Yes	✓ Yes	✓ Yes

Feature	Linux	Windows	Greengrass lite (Linux)
Use the AWS IoT Device SDK for C++ in custom components	✓ Yes	✓ Yes	✓ Yes
Use the AWS IoT Device SDK for Java in custom components	✓ Yes	✓ Yes	✓ Yes

Device certification

Feature	Linux	Windows	Greengrass lite (Linux)
Use AWS IoT Device Tester for AWS IoT Greengrass V2 to validate IoT devices	✓ Yes	✓ Yes	✗ No

Choosing your AWS IoT Greengrass nucleus runtime

As of version 2.14.0, AWS IoT Greengrass provides two alternative implementations of its device runtime, an executable known as the nucleus. Despite their implementation differences, both runtimes are compatible with the AWS IoT Greengrass service and APIs and allow you to deploy components provided by AWS or develop custom components using the Greengrass SDK. It is also possible to mix devices, using either type of nucleus within the same fleet as necessary.

However, in order to achieve the desired portability or the specific memory saving benefits, it is essential to ensure that the nucleus you deploy on your Greengrass devices is compatible with the components you intend to use to accelerate the development of your AWS IoT solutions. To learn more about component compatibility, see Components.

Ultimately, the choice between the two Greengrass runtime options will depend on your specific use case, device constraints, feature requirements, and operating system.

Greengrass nucleus

AWS IoT Greengrass nucleus is the fully-featured runtime that enables you to run AWS IoT Greengrass on a wide range of devices, including gateways, servers, and edge devices with more compute resources. Consider choosing Greengrass nucleus if:

- **Compute resources:** Your device has sufficient compute resources, such as more than 128 MB of RAM and a relatively powerful processor (for example, greater than 1 GHz clock).
- **Full OS support is needed:** Greengrass nucleus supports the widest range of operating systems (including most Linux distros and Windows).
- **Components compatibility:** Greengrass nucleus offers the fullest compatibility with existing components published by the AWS IoT service team and partners.

Greengrass nucleus lite

AWS IoT Greengrass nucleus lite is a lightweight, open-source runtime that enables you to run AWS IoT Greengrass on resource-constrained devices. This can be useful for low-cost, single-board computers with high-volume applications, such as smart home hubs, smart energy meters, smart vehicles, edge AI, and robotics. Consider choosing Greengrass nucleus lite if your devices are:

- **Resource-constrained:** Your device has limited resources, such as RAM memory (512 MB or less), storage (FLASH) space or a low-performance processor (less than 1 GHz).
- **Dependency limited:** Your device vendor software platform does not support Java or the specific JVM required by Greengrass nucleus.
- **Operating system:** Your devices run a distribution of Linux that supports *systemd* (for example: Ubuntu, Yocto).

Current limitations of Greengrass nucleus lite

Greengrass nucleus lite runtime offers a subset of the functionality available in Greengrass nucleus.

The AWS IoT Greengrass IPC (interprocess communication) mechanism allows components to communicate with the Greengrass nucleus. The lightweight version of the nucleus supports the following subset:

Feature	Availability
SubscribeToTopic	Available
PublishToTopic	Available
PublishToIoTCore	Available
SubscribeToIoTCore	Available
UpdateState	Not currently available
SubscribeToComponentUpdates	Not currently available
DeferComponentUpdate	Not currently available
GetConfiguration	Available
UpdateConfiguration	Available
SubscribeToConfigurationUpdate	Available
SubscribeToValidateConfigurationUpdates	Not currently available.
SendConfigurationValidityReport	Not currently available.
GetSecretValue	Not currently available.
PutComponentMetric	Not currently available
GetComponentDetails	Not currently available
RestartComponent	Not currently available
StopComponent	Not currently available
CreateLocalDeployment	Available
CancelLocalDeployment	Not currently available
GetLocalDeploymentStatus	Not currently available

Feature	Availability
ListLocalDeployments	Not currently available
ListComponents	Not currently available
ValidateAuthorizationToken	Available
CreateDebugPassword	Not currently available
PauseComponent	Not currently available
ResumeComponent	Not currently available
GetThingShadow	Not currently available
UpdateThingShadow	Not currently available
DeleteThingShadow	Not currently available
ListNamedShadowsForThing	Not currently available
SubscribeToCertificateUpdates	Not currently available
VerifyClientDeviceIdentity	Not currently available
GetClientDeviceAuthToken	Not currently available
AuthorizeClientDeviceAction	Not currently available

What's new in AWS IoT Greengrass Version 2

AWS IoT Greengrass Version 2 is a major version of AWS IoT Greengrass that introduces the following features:

- **AWS IoT Greengrass nucleus lite** – AWS IoT Greengrass nucleus versions 2.14.0 and later offers AWS IoT Greengrass nucleus lite. Greengrass nucleus lite is a lightweight, open-source runtime that lets you run Greengrass on resource-constrained devices. This can be useful for low-cost, single-board computers with high-volume applications, such as smart home hubs, smart energy meters, smart vehicles, edge AI, and robotics. For more information, see Greengrass nucleus lite.
- **Publisher-supported components** – AWS IoT Greengrass now offers Publisher-supported components. These components are developed, offered, and serviced by third-party vendors. For more information, see Publisher-supported components.
- **Operate a Greengrass device in VPC** – Operating a Greengrass core device in VPC is now available. This enables you to perform deployments in VPC without public internet access. For more information, see Operate an AWS IoT Greengrass core device in VPC.
- **Greengrass Testing Framework (GTF)** – GTF for AWS IoT Greengrass Version 2 is now available. GTF is a collection of building blocks to support end-to-end automation. It enables AWS IoT Greengrass Version 2 internal customers to use the same testing framework that the service team uses for qualifying software changes, automated acceptance, and quality assurance purposes. For more information, see Greengrass Testing Framework on Github.
- **PSA-certified** – AWS IoT Greengrass nucleus versions 2.7.0 and later are now Platform Security Architecture (PSA) certified. For more information, see AWS IoT Greengrass is PSA-certified.

AWS IoT Greengrass release notes provide details about AWS IoT Greengrass releases—new features, updates and improvements, and general fixes. AWS IoT Greengrass has the following types of releases:

- New feature releases for AWS IoT Greengrass
- AWS IoT Greengrass Core software updates

This section contains all of the AWS IoT Greengrass V2 release notes, latest first, and includes major feature changes and significant bug fixes. For information about additional minor fixes, see the aws-greengrass organization on GitHub.

Release notes

- Release: AWS IoT Greengrass Core v2.18.2 software update on July 20, 2026
- Release: AWS IoT Greengrass Core v2.18.1 software update on July 15, 2026
- Release: AWS IoT Greengrass Core v2.18.0 software update on July 8, 2026
- Release: Greengrass nucleus lite v2.5.1 software update on May 6, 2026
- Release: AWS IoT Greengrass Core v2.17.0 software update on April 16, 2026
- Release: AWS IoT Greengrass Core v2.16.1 software update on December 23, 2025
- Release: AWS IoT Greengrass Core v2.16.0 software update on November 6, 2025
- Release: AWS IoT Greengrass Core v2.15.1 software update on November 6, 2025
- Release: AWS IoT Greengrass Core v2.15.0 software update on July 3, 2025
- Release: AWS IoT Greengrass Core v2.14.3 software update on April 11, 2025
- Release: AWS IoT Greengrass Core v2.14.2 software update on March 27, 2025
- Release: AWS IoT Greengrass Core v2.14.1 software update on February 7, 2025
- Release: AWS IoT Greengrass Core v2.14.0 software update on December 16, 2024
- Release: AWS IoT Greengrass Core v2.13.0 software update on August 26, 2024
- Release: AWS IoT Greengrass Core v2.12.6 software update on May 24, 2024
- Release: AWS IoT Greengrass Core v2.12.5 software update on April 25, 2024
- Release: AWS IoT Greengrass Core v2.12.4 software update on April 02, 2024
- Release: AWS IoT Greengrass Core v2.12.3 software update on March 27, 2024
- Release: AWS IoT Greengrass Core v2.12.2 software update on February 15, 2024
- Release: AWS IoT Greengrass Core v2.12.1 software update on December 8, 2023
- Release: AWS IoT Greengrass Core v2.12.0 software update on November 7, 2023
- Release: AWS IoT Greengrass Core v2.11.3 software update on October 18, 2023
- Release: AWS IoT Greengrass Core v2.11.2 software update on August 9, 2023
- Release: AWS IoT Greengrass Core v2.11.1 software update on July 21, 2023
- Release: AWS IoT Greengrass Core v2.11.0 software update on June 28, 2023
- Release: AWS IoT Greengrass Core v2.10.3 software update on June 21, 2023
- Release: AWS IoT Greengrass Core v2.10.2 software update on June 5, 2023
- Release: AWS IoT Greengrass Core v2.10.1 software update on May 11, 2023
- Release: AWS IoT Greengrass Core v2.10.0 software update on May 9, 2023

- Release: AWS IoT Greengrass Core v2.9.6 software update on April 20, 2023
- Release: AWS IoT Greengrass Core v2.9.5 software update on March 30, 2023
- Release: AWS IoT Greengrass Core v2.9.4 software update on February 24, 2023
- Release: AWS IoT Greengrass Core v2.9.3 software update on February 01, 2023
- Release: AWS IoT Greengrass Core v2.9.2 software update on December 22, 2022
- Release: AWS IoT Greengrass Core v2.9.1 software update on November 18, 2022
- Release: AWS IoT Greengrass Core v2.9.0 software update on November 15, 2022
- Release: AWS IoT Greengrass Core v2.8.1 software update on October 13, 2022
- Release: AWS IoT Greengrass Core v2.8.0 software update on October 7, 2022
- Release: AWS IoT Greengrass Core v2.7.0 software update on July 28, 2022
- Release: AWS IoT Greengrass Core v2.6.0 software update on June 27, 2022
- Release: AWS IoT Greengrass Core v2.5.6 software update on May 31, 2022
- Release: AWS IoT Greengrass Core v2.5.5 software update on April 6, 2022
- Release: AWS IoT Greengrass Core v2.5.4 software update on March 23, 2022
- Release: AWS IoT Greengrass Core v2.5.3 software update on January 6, 2022
- Release: AWS IoT Greengrass Core v2.5.2 software update on December 3, 2021
- Release: AWS IoT Greengrass Core v2.5.1 software update on November 23, 2021
- Release: AWS IoT Greengrass Core v2.5.0 software update on November 12, 2021
- Release: AWS IoT Greengrass Core v2.4.0 software update on August 3, 2021
- Release: AWS IoT Greengrass Core v2.3.0 software update on June 29, 2021
- Release: AWS IoT Greengrass Core v2.2.0 software update on June 18, 2021
- Release: AWS IoT Greengrass Core v2.1.0 software update on April 26, 2021
- Release: AWS IoT Greengrass Core v2.0.5 software update on March 09, 2021
- Release: AWS IoT Greengrass Core v2.0.4 software update on February 04, 2021

Release: AWS IoT Greengrass Core v2.18.2 software update on July 20, 2026

This release provides version 2.18.2 of the Greengrass nucleus component and version 2.18.2 of the Greengrass CLI component.

Release date: July 20, 2026

Release details

- Public component updates

Public component updates

The following table lists components provided by AWS that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.18.2 of the Greengrass nucleus is available. Bug fixes and improvements • Fixes an issue where the nucleus could write unchanged configuration values to the configuration store.
Greengrass CLI	Updates the component version for the Greengrass nucleus version 2.18.2 release.

Release: AWS IoT Greengrass Core v2.18.1 software update on July 15, 2026

This release provides version 2.18.1 of the Greengrass nucleus component and version 2.18.1 of the Greengrass CLI component.

Release date: July 15, 2026

Release details

- Public component updates

Public component updates

The following table lists components provided by AWS that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.18.1 of the Greengrass nucleus is available. Bug fixes and improvements • Fixes an issue where the nucleus might fail to fetch credentials after a transient connection failure.

Component	Details
Greengrass CLI	Updates the component version for the Greengrass nucleus version 2.18.1 release.

Release: AWS IoT Greengrass Core v2.18.0 software update on July 8, 2026

This release provides version 2.18.0 of the Greengrass nucleus component and updates to AWS-provided components.

Release date: July 8, 2026

Release details

- Public component updates

Public component updates

The following table lists components provided by AWS that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
>
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.18.0 of the Greengrass nucleus is available. New features • Adds support for switching a core device's AWS IoT connectivity to a different AWS account or region through a nucleus configuration deployment. The nucleus validates connectivity to the new endpoints before it applies the change, and reports the terminal deployment status to the source account. • Adds support for Windows Server 2025. Bug fixes and improvements • Fixes a startup timing issue on systemd devices by starting the AWS IoT Greengrass Core software only after the system clock synchronizes. • Fixes an issue where a component might encounter stale subscript ions after it closes a local IPC stream during an in-progress subscribe request. • General bug fixes and improvements.
Stream manager	Version 2.3.1 of the stream manager is available. Bug fixes and improvements • Improves the throughput of stream manager exports to Kinesis Data Streams destinations, especially for small record sizes. • Adds shard-aware pacing that distributes records across the shards of the target data stream to keep requests within the per-shard limits of Kinesis Data Streams. To use shard-aware pacing, grant the `kinesis:ListShards` permission in the Greengrass token exchange role. • Adds the `STREAM_MANAGER_EXPORTER_KINESIS_INTE R_CALL_DELAY_MS` configuration parameter, which sets a fallback delay between export requests when stream manager cannot determine the shard layout of the target data stream. The default is 1000 milliseconds. To turn off the fallback delay, set this parameter to `0`.

Component	Details
Shadow manager	Version 2.3.15 of the shadow manager is available. Bug fixes and improvements • Fixes an issue where the shadow manager could stop working after a power outage corrupted its local database. • Validates the integrity of the local database on startup and recreates the database when it detects corruption.
MQTT bridge	Version 2.3.3 of the MQTT bridge is available. Bug fixes and improvements • Fixes an issue where high concurrent device load could cause the MQTT bridge to stop relaying messages after a dropped PUBACK.
Secret manager	Version 2.2.9 of the secret manager is available. Bug fixes and improvements • General bug fixes and improvements.
Greengrass CLI	Version 2.18.0 of the Greengrass CLI is available. Updates the component version for the Greengrass nucleus version 2.18.0 release.
Secure tunneling	Version 2.0.1 of the secure tunneling component is available. Bug fixes and improvements • Updates the bundled local proxy binary to version 3.3.0. • Fixes concurrency and object-lifetime bugs that could affect tunnel data integrity. • Requires TLS 1.2 or later and no longer allows SSLv2, SSLv3, TLS 1.0, or TLS 1.1.

Release: Greengrass nucleus lite v2.5.1 software update on May 6, 2026

This release provides version 2.5.1 of the Greengrass nucleus lite component.

Release date: May 6, 2026

Release details

- Public component updates

Public component updates

The following table lists components provided by AWS that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus lite Version 2.5.1	Bug fixes and improvements • Fixes an issue where `SubscribeToConfigurationUpdate` could notify subscribers when a configuration value did not actually change. • Fixes an issue where the deployment source ARN of components could be overwritten by unrelated deployments.

Component	Details
	• Fixes an issue where HTTP artifact download retries did not trigger on retryable error codes such as 5xx and 429.

Release: AWS IoT Greengrass Core v2.17.0 software update on April 16, 2026

This release provides version 2.17.0 of the Greengrass nucleus component, version 2.5.0 of the Greengrass nucleus lite component, and updates to AWS-provided components.

Release date: April 16, 2026

Release details

- Public component updates

Public component updates

The following table lists components provided by AWS that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
>
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.17.0 of the Greengrass nucleus is available. New features • Installs all Amazon Root CA certificates during auto-provisioning. • Adds an uninstall lifecycle that runs when a deployment removes a component. • Allows installing Greengrass Core software as a regular Linux user.
Greengrass nucleus lite	Version 2.5.0 of the Greengrass nucleus lite is available. New features • Adds PKCS#11 support via OpenSSL Providers. • Adds support for using TPM with fleet provisioning.
Stream manager	Version 2.3.0 of the stream manager is available. Bug fixes and improvements • Updates to the AWS SDK for Java v2 for improved performance and compatibility. • Changes proxy precedence behavior to align with standard AWS SDK proxy configuration.
Greengrass CLI	Version 2.17.0 of the Greengrass CLI is available. Updates the component version for the Greengrass nucleus v2.17.0 release.
Secure tunneling	Version 2.0.0 of the secure tunneling component is available. New features • Replaces the Java wrapper with a C wrapper. • Replaces AWS IoT Device Client with AWS IoT Securetunneling Localproxy. • Reduces the binary size from approximately 36 MB to approximately 4 MB.

Component	Details
	• Reduces the memory footprint from approximately 100 MB to approximately 2 MB.

Release: AWS IoT Greengrass Core v2.16.1 software update on December 23, 2025

This release provides version 2.16.1 of the Greengrass nucleus component and version 2.16.1 of the Greengrass CLI component.

Release date: December 23, 2025

Public component updates

The following table lists components provided by AWS that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.16.1 of the Greengrass nucleus is available.

Component	Details
	Bug fixes and improvements • Adds configuration for credential retry intervals after Token Exchange Service failures.
Greengrass CLI	Version 2.16.1 of the Greengrass CLI is available. Version updated for the Greengrass Nucleus v2.16.1 release.

Release: AWS IoT Greengrass Core v2.16.0 software update on November 6, 2025

This release provides version 2.16.0 of the Greengrass nucleus component and updates to AWS-provided components.

Release date: November 6, 2025

Release details

- Public component updates

Public component updates

The following table lists components provided by AWS that include new and updated features.

Important

When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you

create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.16.0 of the Greengrass nucleus is available. New features • Nucleus now supports cgroups V2. Bug fixes and improvements • Fixes an issue where Nucleus would not clean up stale deployments when a deployment was cancelled.
Greengrass nucleus lite	Version 2.3.0 of the Greengrass nucleus lite is available. New features • Support for using TPM 2.0 for IoT Core MQTT authorization. • The sample apt packages now support more operating systems: Ubuntu 22.04, Ubuntu 24.04, Debian 12, and Debian 13. • RestartComponent IPC is now supported. Bug fixes and improvements • Local deployments no longer need internet access. • GetConfiguration has been updated to match the Greengrass Nucleus runtime behavior. (**Breaking Change**) • General bug fixes and improvements.
Shadow manager	Version 2.3.12 of the shadow manager is available. Bug fixes and improvements • Fixes an issue where deployments were blocked when more than 1024 device shadows were configured.
Log manager	Version 2.3.11 of the log manager is available.

Component	Details
	Bug fixes and improvements • Fixes an issue where Log Manager runtime configuration grew indefinit ely with stale information of uploaded log files.
System log forwarder	Version 2.1.0 of the system log forwarder is available. Bug fixes and improvements • Updates the component recipe to properly support Greengrass nucleus. • Improved logging output when there are no logs to upload. • General bug fixes and improvements.

Release: AWS IoT Greengrass Core v2.15.1 software update on November 6, 2025

This release provides version 2.15.1 of the Greengrass nucleus component and version 2.15.1 of the Greengrass CLI component.

Release date: November 6, 2025

Release details

- Public component updates

Public component updates

The following table lists components provided by AWS that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those

devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus Version 2.15.1	Bug fixes and improvements • Fixes an issue where Nucleus failed to clean up component processes after the 30-second shutdown timeout during bootstrap deployments.
Greengrass CLI Version 2.15.1	Version updated for the Greengrass Nucleus v2.15.1 release.

Release: AWS IoT Greengrass Core v2.15.0 software update on July 3, 2025

This release provides version 2.15.0 of the Greengrass nucleus component and updates to AWS-provided components.

Release date: July 3, 2025

Release details

- Public component updates

Public component updates

The following table lists components provided by AWS that include new and updated features.

Important

When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.15.0 of the Greengrass nucleus is available. New features • Adds telemetry feature to include host system information like CPU and OS details. For more information, see telemetry metrics. • Adds `deploymentConfigurationTimeSource` configuration. For more information, see Greengrass nucleus configuration. Bug fixes and improvements • Nucleus now prioritizes the use of local versions and artifacts by default, improving deployment consistency and reducing external dependencies. Users can override this behavior by specifying alternati ve preferences in the deployment document. • Adds a new warning log when a provided Windows user doesn't match the character set normally accepted by Windows.
Greengrass CLI	Version 2.15.0 of the Greengrass CLI is available.

Component	Details
	Bug fixes and improvements • Adds a fix so that when the auth configuration changes, the component won't reinstall but will still restart. • General bug fixes and improvements.
Greengrass nucleus lite	Version 2.2.0 of the Greengrass nucleus lite is available. New features • Adds support for container image artifact URIs. Bug fixes and improvements • General bug fixes and improvements.
Log manager	Version 2.3.10 of the log manager is available. New features • Adds a new configuration key (`updateToTlogIntervalSec`) to control the frequency at which log-upload event details are persisted to the local transaction log (`config.tlog`). Bug fixes and improvements • Improves log manager to refresh cloudwatch client for socket connection error.
Secret manager	Version 2.2.6 of the secret manager is available. Bug fixes and improvements • Fixes an issue where secret manager fails to get a secret due to a slow or unresponsive Trusted Platform Module.

Release: AWS IoT Greengrass Core v2.14.3 software update on April 11, 2025

This release provides version 2.14.3 of the Greengrass nucleus component and updates to AWS-provided components.

Release date: April 11, 2025

Release details

- Public component updates

Public component updates

The following table lists components provided by AWS that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.14.3 of the Greengrass nucleus is available. Bug fixes and improvements • Allows the token exchange service to restart on port configuration changes. • Fixes an issue where fleet status service doesn't send component status change messages for unpinned lambdas if they aren't triggered. • Fixes an issue where components aren't gracefully shutdown when a new component version is deployed. • Fixes an issue where built-in plugins' injection hooks are executed twice resulting in extra lifecycle events and duplicate logging.

Component	Details
	• Improves component lifecycle logging on Windows devices.
Greengrass CLI	Version 2.14.3 of the Greengrass CLI is available. Bug fixes and improvements • Version updated for Greengrass nucleus version 2.14.3 release.
Secret manager	Version 2.2.4 of the secret manager is available. Bug fixes and improvements • Reduces the frequency of writes to the local secret store. Secret manager now writes to the local store only when secrets are updated.
EMQX MQTT broker	Version 2.0.3 of the EMQX MQTT broker is available. Bug fixes and improvements • Fixes an issue where EMQX doesn't start on Windows if the path contains spaces.

Release: AWS IoT Greengrass Core v2.14.2 software update on March 27, 2025

This release provides version 2.14.2 of the Greengrass nucleus component and updates to AWS-provided components.

Release date: March 27, 2025

Release details

- Public component updates

Public component updates

The following table lists components provided by AWS that include new and updated features.

Important

When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.14.2 of the Greengrass nucleus is available. Bug fixes and improvements • Fixes an issue where an HTTP client isn't configured with mutual auth.
Greengrass CLI	Version 2.14.2 of the Greengrass CLI is available. Bug fixes and improvements • Version updated for Greengrass nucleus version 2.14.2 release.

Release: AWS IoT Greengrass Core v2.14.1 software update on February 7, 2025

This release provides version 2.14.1 of the Greengrass nucleus component and updates to AWS-provided components.

Release date: February 7, 2025

Release details

- Public component updates

Public component updates

The following table lists components provided by AWS that include new and updated features.

Important

When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.14.1 of the Greengrass nucleus is available. Bug fixes and improvements • Fixes an issue where components weren't stopped properly on new installations of Greengrass.
Greengrass CLI	Version 2.14.1 of the Greengrass CLI is available. Bug fixes and improvements • Version updated for Greengrass nucleus version 2.14.1 release.
Secure tunneling	Version 1.1.1 of the secure tunneling component is available. Bug fixes and improvements • Adds a configuration to support Greengrass nucleus lite.

Release: AWS IoT Greengrass Core v2.14.0 software update on December 16, 2024

This release provides version 2.14.0 of the Greengrass nucleus component, and new AWS IoT Greengrass nucleus lite updates. The AWS IoT Greengrass nucleus lite is a new runtime, available for AWS IoT Greengrass version 2. It provides a reduced memory footprint alternative. This is a good option for resource-constrained devices. It implements a subset of the nucleus functionality with increased featured compatibility planned for future releases. The source code is available now on Github. With the nucleus lite runtime you can:

- Deploy components to Greengrass core devices. Use the same recipe format, though some advanced features may not be available yet.
- Applications deployed as Greengrass components can use the device SDKs to access the supported Greengrass IPC APIs, such as: AWS IoT Core MQTT access, local pub/sub, and Greengrass configuration access. See the compatibility chart for the list of supported IPC APIs.
- Some AWS managed components have been updated for nucleus lite support. See the AWS-provided components for a list of existing compatible components.

New features:

- Uses less memory and disk space (less than 5MB of RAM and less than 5MB of storage).
- Components integrate with the host system's service manager (systemd for currently supported Linux platforms).

Things to watch out for:

- AWS IoT Greengrass nucleus lite recipes are case-sensitive. Ensure the correct (keys) casing is used as in the https://docs.aws.amazon.com/greengrass/v2/developerguide/component-recipe-reference.htmlrecipe reference.
- The nucleus lite runtime supports **thing group** deployments, and does not yet support the (single) **Core device** deployment target type. To deploy to a single Greengrass device, use a thing group with only that one device in it.
- The nucleus lite runtime uses bounded memory resources; functionality which scales according to usage on the classic runtime may fail due to exceeding resources available on lite. This includes a current limitation on max of 50 MQTT subscriptions at a time, and maximum limits

on recipe file sizes and deployments. Some of these limits are configurable at compile time if compiling the lite runtime yourself.

- The nucleus lite runtime does not ship with Java. To use components requiring Java, the system will need Java already installed, or a component may be used to install Java.
- We recommend compiling the nucleus lite runtime from source and using your own build tailored for your system. For Yocto systems, a layer is available to integrate the nucleus lite runtime into your system image.
- Currently the nucleus lite assumes a Linux system using *systemd*, or a container image using *systemd*.
- While you can manage Docker containers with recipe scripts, Greengrass managed container artifacts are not yet available.
- The nucleus lite runtime does not yet have support for keys stored in a PKCS11 module. If your use case requires keys stored on a secure element, the classic runtime can support this use case currently. To prevent leaks of your device credentials, ensure production devices are using full disk encryption.

Alongside the introduction of nucleus lite, we are also releasing nucleus v2.14.0. This update brings significant enhancements to the existing Greengrass nucleus.

Key features and improvements:

- New dual-stack endpoint support enables IPv6 network communication.
- Enhanced resilience against nucleus restart failures and directory corruption.
- Fixed memory leaks in IPC PubSub subscription closures.

Release date: December 16, 2024

Public component updates

The following table lists AWS-provided components that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-

provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus lite	Version 2.0.0 of the Greengrass nucleus lite is available. New features • Uses less memory and disk space (less than 5MB of RAM and less than 5MB of storage). • Components integrate with the host system's service manager (systemd for currently supported Linux platforms).
Greengrass nucleus	Version 2.14.0 of the Greengrass nucleus is available. New features • New dual-stack endpoint support enables IPv6 network communica tion. • Enhanced resilience against nucleus restart failures and directory corruption. Bug fixes and improvements • Fixed memory leaks in IPC PubSub subscription closures. • Fixes run lifecycle of the component where it enters into ERRORED state due to startup timeout when skipif condition is true. • Fixes an issue where the core device fails to connect to AWS IoT Core when the TLS policy is set to TLS13_1_3_2022_10.
Greengrass CLI	Version 2.14.0 of the Greengrass CLI is available.

Component	Details
	Bug fixes and improvements • Validate deployment target parameter in the cli command.
Stream manager	Version 2.14.0 of the Stream manager is available. New features • Adds a new configuration key for startup timeout. Default value is 120 seconds. • Add recipe supports for Greengrass nucleus lite.
MQTT 5 broker (EMQX)	Version 2.0.2 of the MQTT 5 broker (EMQX) is available. Bug fixes and improvements • Fixes an issue where EMQX starts up before the Client device auth component is ready.
Lambda runtimes component	Version 2.0.9 of the Lambda runtimes component is available. Bug fixes and improvements • Fixes an syntax warning with Python 3.12
Lambda manager component	Version 2.3.5 of the Lambda manager component is available. Bug fixes and improvements • Improves performance by using epoll instead of nio when available
Secret manager component	Version 2.2.2 of the Secret manager component is available. Bug fixes and improvements • Fixes an issue where secret manager doesn't download the secrets configured with partial arns.
Secure tunneling component	Version 1.1.0 of the Secure tunneling component is available. New features • Add recipe supports for Greengrass nucleus lite

Component	Details
CloudWatch metrics component	Version 1.1.0 of the CloudWatch metrics component is available. New features • Add recipe supports for Greengrass nucleus lite

Release: AWS IoT Greengrass Core v2.13.0 software update on August 26, 2024

This release provides version 2.13.0 of the Greengrass nucleus component.

Release date: August 26, 2024

Public component updates

The following table lists AWS-provided components that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.13.0 of the Greengrass nucleus is available.

Component	Details
	New features • Support FIPS endpoint in Nucleus. For more information, see FIPS endpoints. Bug fixes and improvements • Cancel deployment improvements - deployments can now be cancelled while new configuration is being merged and while waiting for services to start.
Stream manager	Version 2.1.13 of the Stream manager component is available. Bug fixes and improvements • Support FIPS endpoint in AWS IoT SiteWise
Secret manager	Version 2.2.0 of the Secret manager component is available. New features • Adds support for periodic refresh of configured secrets through a new component configuration key. • Adds support for a new request parameter in the **GetSecretValue** IPC request to refresh the secrets per request
IP detector	Version 2.2.0 of the IP detector component is available. New features • Adds support for IPv6. You can now use IPv6 for local messaging.
Client device auth	Version 2.5.1 of the Client device auth is available. Bug fixes and improvements • General bugs and fixes. • Supports FIPS endpoint.

Component	Details
Local debug console	Version 2.4.3 of the Local debug console is available. Bug fixes and improvements • Fixes an issue that incorrectly displayed STREAM_MANAGER_EXPORTER_MAX_BANDWIDTH in Mpbs instead of bytes/sec.

Release: AWS IoT Greengrass Core v2.12.6 software update on May 24, 2024

This release provides version 2.12.6 of the Greengrass nucleus component and updates to AWS-provided components.

Release date: May 24, 2024

Release details

- Public component updates

Public component updates

The following table lists components provided by AWS that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.12.6 of the Greengrass nucleus is available. Bug fixes and improvements • Fixes an issue that causes a crash at startup on certain ARMv8 processors, including the Jetson Nano.
Greengrass CLI	Version 2.12.6 of the Greengrass CLI is available. Bug fixes and improvements • Version updated for Greengrass nucleus version 2.12.6 release.
Secret manager	Version 2.1.8 of the secret manager is available. Bug fixes and improvements • Fixes an issue where secret manager doesn't accept a partial arn.

Release: AWS IoT Greengrass Core v2.12.5 software update on April 25, 2024

This release provides version 2.12.5 of the Greengrass nucleus component and updates to AWS-provided components.

Release date: April 25, 2024

Release details

- Public component updates

Public component updates

The following table lists components provided by AWS that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-

provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.12.5 of the Greengrass nucleus is available. Bug fixes and improvements • Fixes an issue where deployment rollback occasionally gets stuck while rolling back a previously broken component with hard dependencies. • Fixes an issue where the nucleus doesn't publish status updates after fleet provisioning. • Adds retries for the `GetDeploymentConfiguration` API after getting 404 errors.

Release: AWS IoT Greengrass Core v2.12.4 software update on April 02, 2024

This release provides version 2.12.4 of the Greengrass nucleus component and updates to AWS-provided components.

Release date: April 02, 2024

Release details

- Public component updates

Public component updates

The following table lists components provided by AWS that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.12.4 of the Greengrass nucleus is available. Bug fixes and improvements • Fixes an issue where the nucleus enters a deadlock condition during startup on some Linux devices.

Release: AWS IoT Greengrass Core v2.12.3 software update on March 27, 2024

This release provides version 2.12.3 of the Greengrass nucleus component and updates to AWS-provided components.

Release date: March 27, 2024

Release details

- Public component updates

Public component updates

The following table lists components provided by AWS that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
>
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.12.3 of the Greengrass nucleus is available. Bug fixes and improvements • Fixes an issue where the nucleus doesn't report the correct component status after the nucleus relaunches and during component recovery. • General bug fixes and improvements.
Shadow manager	Version 2.3.7 of the shadow manager component is available. Bug fixes and improvements Fixes an issue where shadow manager periodically logs a `NullPointerException` error during a shadow manager sync.
Fleet provisioning	Version 1.2.1 of the AWS IoT fleet provisioning plugin is available.

Component	Details
	Bug fixes and improvements Fixes an issue where the fleet provisioning plugin is offline during a Greengrass nucleus startup. The fleet provisioning plugin now indefinit ely retries MQTT connect calls.
IP detector	Version 2.1.9 of the disk spooler component is available. Bug fixes and improvements Adjusts the IP acquired step to only send logs at the debug log level.
Moquette MQTT 3.1.1 broker component	Version 2.3.6 of the Moquette MQTT 3.1.1 broker component is available. Bug fixes and improvements General bug fixes and improvements.
Lambda manager	Version 2.3.3 of the Lambda manager component is available. Bug fixes and improvements General bug fixes and improvements.
Local debug console	Version 2.4.2 of the local debug console component is available. Bug fixes and improvements General bug fixes and improvements.

Release: AWS IoT Greengrass Core v2.12.2 software update on February 15, 2024

This release provides version 2.12.2 of the Greengrass nucleus component and updates to AWS-provided components.

Release date: February 15, 2024

Release details

- Public component updates

Public component updates

The following table lists components provided by AWS that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.12.2 of the Greengrass nucleus is available. Bug fixes and improvements • Fixes an issue where old logs weren't cleaned up properly. • General bug fixes and improvements.
Shadow manager	Version 2.3.6 of the shadow manager component is available. Bug fixes and improvements Fixes an issue where shadow properties that are deleted through AWS Cloud updates while the device is offline continue to exist in the local shadow after regaining connectivity.
Lambda launcher	Version 2.0.13 of the lambda launcher component is available.

Component	Details
	Bug fixes and improvements General bug fixes and improvements.
Disk spooler	Version 1.0.3 of the disk spooler component is available. Bug fixes and improvements Improves performance by reusing database connections.

Release: AWS IoT Greengrass Core v2.12.1 software update on December 8, 2023

This release provides version 2.12.1 of the Greengrass nucleus component and updates to AWS-provided components.

Release date: December 8, 2023

Release details

- Public component updates

Public component updates

The following table lists components provided by AWS that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you

create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.12.1 of the Greengrass nucleus is available. Bug fixes and improvements • Fixes an issue where the nucleus may duplicate MQTT subscriptions to deployment topics leading to additional logging and MQTT publishes.
Client device auth	Version 2.4.5 of the client device auth component is available. New features Adds support for wildcard prefixes for selecting thing names with the `selectionRule` parameter. Bug fixes and improvements Fixes an issue where certificates aren't updated with new connectivity information in certain cases.
Disk spooler	Version 1.0.2 of the disk spooler component is available. Bug fixes and improvements Fixes an issue where the MQTT message format field isn't persisted in certain cases.
MQTT bridge	Version 2.3.1 of the disk spooler component is available. Bug fixes and improvements Fixes an issue where the local MQTT client gets into a disconnect loop.
Stream manager	Version 2.1.12 of the stream manager component is available.

Component	Details
	Bug fixes and improvements Updates the order that credentials are used so that Greengrass credentials are preferred for AWS service requests.

Release: AWS IoT Greengrass Core v2.12.0 software update on November 7, 2023

This release provides version 2.12.0 of the Greengrass nucleus component and updates to AWS-provided components.

Release date: November 7, 2023

Release highlights

- **Bootstrap on rollback** – AWS IoT Greengrass now provides a Greengrass nucleus configuration parameter called `BootstrapOnRollback`. This feature enables you to run the bootstrap lifecycle steps as part of a rollback deployment.

Release details

- Public component updates

Public component updates

The following table lists components provided by AWS that include new and updated features.

Important

When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.

To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.12.0 of the Greengrass nucleus is available. New features • Enables you to run the bootstrap lifecycle steps as part of a rollback deployment.

Release: AWS IoT Greengrass Core v2.11.3 software update on October 18, 2023

This release provides version 2.11.3 of the Greengrass nucleus component.

Release date: October 18, 2023

Release details

- Public component updates

Public component updates

The following table lists components provided by AWS that include new and updated features.

Important

When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.

To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.11.3 of the Greengrass nucleus is available. Bug fixes and improvements • Fixes an issue in the nucleus where it may improperly start a component when its dependencies fail. New features • Adds configurable s3 endpoint type.
Lambda manager	Version 2.3.1 of the Lambda manager component is available. Bug fixes and improvements • Adjusts log levels for certain errors.
Local deubg console	Version 2.4.0 of the Lambda manager component is available. New features • Adds stream manager debugging console.
Log manager	Version 2.3.6 of the log manager component is available. Bug fixes and improvements • Adjusts log levels for certain errors.
Shadow manager	Version 2.3.4 of the Shadow manager component is available. Bug fixes and improvements • Adds support for null and empty shadow state documents.

Release: AWS IoT Greengrass Core v2.11.2 software update on August 9, 2023

This release provides version 2.11.2 of the Greengrass nucleus component.

Release date: August 9, 2023

Release details

- Public component updates

Public component updates

The following table lists components provided by AWS that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.11.2 of the Greengrass nucleus is available. Bug fixes and improvements • Fixes an issue in the nucleus MQTT 5 client where it may appear offline when a large number (> 50) of subscriptions are in use. • Adds a retry for the docker dial TCP failure.

Release: AWS IoT Greengrass Core v2.11.1 software update on July 21, 2023

This release provides version 2.11.1 of the Greengrass nucleus component.

Release date: July 21, 2023

Release details

- Public component updates

Public component updates

The following table lists components provided by AWS that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.11.1 of the Greengrass nucleus is available. Bug fixes and improvements • Fixes an issue where the nucleus doesn't start if a bootstrap task fails and the deployment metadata file is corrupted. • Fixes an issue where on-demand Lambda components aren't reported in deployment status updates.

Component	Details
	• Adds support for duplicate authorization policy IDs.
Lambda manager	Version 2.2.11 of the Lambda manager is available. Bug fixes and improvements • Fixes an issue where the LegacySubscriptionRouter configuration does not update when the Lambda configuration changes.

Release: AWS IoT Greengrass Core v2.11.0 software update on June 28, 2023

This release provides version 2.11.0 of the Greengrass nucleus component.

Release date: June 28, 2023

Release highlights

- **Persistent disk spooler** – AWS IoT Greengrass now provides a persistent spooler implementation for messages spooled from Greengrass core devices to AWS IoT Core. This component will store these outbound messages on disk. For more information, see Disk spooler.
- **Local deployment improvements** – You can now cancel local deployments, set deployment failing handling polices, and get detailed deployment status.
- **Logging speed improvements** – Log upload speeds for the log manager component have been improved.

Release details

- Public component updates

Public component updates

The following table lists components provided by AWS that include new and updated features.

Important

When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.11.0 of the Greengrass nucleus is available. New features • Enables you to cancel a local deployment. • Enables you to configure a failure handling policy for a local deployment. • Adds support for a disk spooler plugin.
Greengrass CLI	Version 2.11.0 of the Greengrass CLI is available. New features • Enables you to cancel a local deployment. • Enables you to configure a failure handling policy for a local deployment. • Improves detailed deployment status reporting.
Disk spooler	Version 1.0.0 of the disk spooler component is available. • The disk spooler component provides persistent storage of messages sent from Greengrass core devices to AWS IoT Core.

Component	Details
Log manager	Version 2.3.5 of the log manager component is available. Improvements Improves log upload speed.

Release: AWS IoT Greengrass Core v2.10.3 software update on June 21, 2023

This release provides version 2.10.3 of the Greengrass nucleus component.

Release date: June 21, 2023

Release details

- Public component updates

Public component updates

The following table lists components provided by AWS that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.10.3 of the Greengrass nucleus is available. Bug fixes and improvements • Fixes an issue where Greengrass doesn't subscribe to deployment notifications when using the PKCS#11 provider.

Release: AWS IoT Greengrass Core v2.10.2 software update on June 5, 2023

This release provides version 2.10.2 of the Greengrass nucleus component.

Release date: June 5, 2023

Release details

- Public component updates

Public component updates

The following table lists components provided by AWS that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.10.2 of the Greengrass nucleus is available. Bug fixes and improvements • Allows case insensitive parsing of component lifecycles. • Fixes an issue where the environment PATH variable was not recreated correctly. • Fixes proxy URI encoding for components including stream manager for usernames with special characters.
Client device auth	Version 2.4.2 of the client device auth component is available. New features Adds a new `startupTimeoutSeconds` configuration option.
Lambda manager	Version 2.2.9 of the Lambda manager component is available. Bug fixes and improvements Fixes an issue where the port number is corrupted due to a skewed clock.
Log manager	Version 2.3.4 of the log manager component is available. Bug fixes and improvements • Adds support for setting the `periodicUploadIntervalSec` parameter to fractional values. The minimum is 1 microsecond. • Fixes an issue where log manager doesn't respect the CloudWatch `putLogEvents` limits.
MQTT 3.1 broker (Moquette)	Version 2.3.3 of the MQTT 3.1 broker (Moquette) component is available. New features Adds a new `startupTimeoutSeconds` configuration option.
MQTT bridge	Version 2.2.6 of the MQTT bridge component is available.

Component	Details
	New features Adds a new `startupTimeoutSeconds` configuration option.
Stream manager	Version 2.1.7 of the stream manager component is available. Bug fixes and improvements Fixes an issue where stream manager fails to read the proxy configuration correctly.

Release: AWS IoT Greengrass Core v2.10.1 software update on May 11, 2023

This release provides version 2.10.1 of the Greengrass nucleus component.

Release date: May 11, 2023

Release details

- Public component updates

Public component updates

The following table lists components provided by AWS that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you

create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.10.1 of the Greengrass nucleus is available. Bug fixes and improvements • Fixes an issue that could cause a crash at startup on certain ARMv8 processors, including the Jetson Nano. • Greengrass no longer closes a component's standard in, this reverts the behavior to the pre-2.10.0 behavior
Stream manager	Version 2.1.6 of the new stream manager is available. Bug fixes and improvements Fixes an issue that could cause a crash at startup on certain ARMv8 processors, including the Jetson Nano.

Release: AWS IoT Greengrass Core v2.10.0 software update on May 9, 2023

This release provides version 2.10.0 of the Greengrass nucleus component and updates to AWS-provided components.

Release date: May 9, 2023

Release highlights

- **MQTT5 support** – AWS IoT Greengrass now supports sending and receiving messages from AWS IoT Core using MQTT5. For more information, see Publish AWS IoT Core MQTT messages.

Release details

- Public component updates

Public component updates

The following table lists components provided by AWS that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.10.0 of the Greengrass nucleus is available. New features • Adds `interpolateComponentConfiguration` support for the empty regular expression. Greengrass now interpolates from the root config object. • Adds support for MQTT5. • Adds a mechanism for loading plugin components quickly without scanning. • Enables Greengrass to save disk space by deleting unused Docker images. Bug fixes and improvements • Fixes an issue where rollback leaves certain configuration values in place from a deployment. • Fixes an issue where the Greengrass nucleus validates for an AWS domain sequence in custom non-AWS credentials and data endpoints.

Component	Details
	• Updates multi-group dependency resolution to re-resolve all group dependencies via AWS Cloud negotiation, instead of locking to the active version. This update also removes the deployment error code `INSTALLED_COMPONENT_NOT_FOUND` . • Updates the Greengrass nucleus to skip downloading Docker images when they already exist locally. • Updates the Greengrass nucleus to restart a component install step before timeout expires. • Additional minor fixes and improvements.
Shadow manager	Version 2.3.2 of the new shadow manager is available. Bug fixes and improvements Fixes an issue where shadow manager enters the BROKEN state when the local shadow database is corrupted.

Release: AWS IoT Greengrass Core v2.9.6 software update on April 20, 2023

This release provides version 2.9.6 of the Greengrass nucleus component.

Release date: April 20, 2023

Release details

- Public component updates

Public component updates

The following table lists components provided by AWS that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-

provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.9.6 of the Greengrass nucleus is available. Bug fixes and improvements • Fixes an issue where a Greengrass deployment fails with the error LAUNCH_DIRECTORY_CORRUPTED and a subsequent device reboot fails to start Greengrass. This error may occur when you move the Greengrass device between multiple thing groups with deployments that require Greengrass to restart.

Release: AWS IoT Greengrass Core v2.9.5 software update on March 30, 2023

This release provides version 2.9.5 of the Greengrass nucleus component.

Release date: March 30, 2023

Release details

- Public component updates

Public component updates

The following table lists components provided by AWS that include new and updated features.

Important

When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.9.5 of the Greengrass nucleus is available. New features • Adds support for Greengrass nucleus software signature verification. Bug fixes and improvements • Fixes an issue where a deployment fails when the local recipe metadata region doesn't match the Greengrass nucleus launch region. The Greengrass nucleus now renegotiates with the cloud when this happens. • Fixes an issue where the MQTT message spooler fills up and never removes messages. • Additional minor fixes and improvements.

Release: AWS IoT Greengrass Core v2.9.4 software update on February 24, 2023

This release provides version 2.9.4 of the Greengrass nucleus component.

Release date: February 24, 2023

Release details

- Public component updates

Public component updates

The following table lists components provided by AWS that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.9.4 of the Greengrass nucleus is available. Bug fixes and improvements • Checks for a null message before it drops QOS 0 messages. • Truncates job status detail values if they exceed the 1024 character limit. • Updates the bootstrap script for Windows to correctly read the Greengrass root path if that path includes spaces. • Updates subscribing to AWS IoT Core so that it drops client messages if the subscription response wasn't sent. • Ensures that the nucleus loads its configuration from backup files when the main configuration file is corrupt or missing.

Release: AWS IoT Greengrass Core v2.9.3 software update on February 01, 2023

This release provides version 2.9.3 of the Greengrass nucleus component.

Release date: February 01, 2023

Release details

- Public component updates

Public component updates

The following table lists components provided by AWS that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.9.3 of the Greengrass nucleus is available. Bug fixes and improvements • Ensures MQTT client IDs aren't duplicated. • Adds more robust file-reading and writing to avoid and recover from corruption.

Component	Details
	• Retries docker image pull on specific network-related errors. • Adds the `noProxyAddresses` option for MQTT connection.

Release: AWS IoT Greengrass Core v2.9.2 software update on December 22, 2022

This release provides version 2.9.2 of the Greengrass nucleus component.

Release date: December 22, 2022

Release details

- Public component updates

Public component updates

The following table lists AWS-provided components that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.9.2 of the Greengrass nucleus is available. Bug fixes and improvements • Fixes an issue where configuring `interpolateComponentConfiguration` doesn't apply to an ongoing deployment. • Uses OSHI to list all child processes.

Release: AWS IoT Greengrass Core v2.9.1 software update on November 18, 2022

This release provides version 2.9.1 of the Greengrass nucleus component and updates to AWS-provided components.

Release date: November 18, 2022

Release highlights

- **Log manager** – Log manager now processes and directly uploads active log files instead of waiting for new files to be rotated. This improvement significantly reduces log delays. For more information, see Log manager

Release details

- Public component updates

Public component updates

The following table lists AWS-provided components that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if

you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.9.1 of the Greengrass nucleus is available. Bug fixes and improvements • Adds fix where Greengrass restarts if a deployment removes a plugin component.
Log manager	Version 2.3.0 of the new log manager is available. **Note** We recommend that you upgrade to Greengrass nucleus 2.9.1 when you upgrade to log manager 2.3.0. New features • Reduces log delays by processing and directly uploading active log files instead of waiting for new files to be rotated. Bug fixes and improvements • Improves support of log rotation when rotating files with a unique name. • Additional minor fixes and improvements.

Release: AWS IoT Greengrass Core v2.9.0 software update on November 15, 2022

This release provides version 2.9.0 of the Greengrass nucleus component and updates to AWS-provided components.

Release date: November 15, 2022

Release highlights

- **Offline authentication** – AWS IoT Greengrass now supports offline authentication. You can configure your AWS IoT Greengrass core device so that client devices can connect to a core device, even when the core device isn't connected to the cloud. For more information, see Offline authentication.
- **Subdeployments** – You can now create subdeployments. You can use a subdeployment to resolve unsuccessful deployments. Each subdeployment can test a different configuration of an unsuccessful deployment on a smaller subset of devices. For more information, see Create subdeployments.

Release details

- Public component updates

Public component updates

The following table lists AWS-provided components that include new and updated features.

Important

When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you

create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.9.0 of the Greengrass nucleus is available. New features • Adds the ability to create subdeployments that retry deployments with a smaller subset of devices. This feature creates a more efficient way to test and resolve unsuccessful deployments. Bug fixes and improvements • Improves support for systems that don't have `useradd`, `groupadd`, and `usermod`. • Additional minor fixes and improvements.
Client device auth	Version 2.3.0 of the client device auth component is available. New features • Adds support for offline authentication of client devices. With this feature, client devices can continue to connect to the core device when the core device isn't connected to the Internet. • Adds support for customer-provided certificate authorities (CA). Your core device uses a customer-provided CA as the root certificate to generate MQTT broker certificates.
MQTT 5 broker (EMQX)	Version 1.2.0 of the MQTT 5 broker (EMQX) component is available. New features Adds support for certificate chains.
Moquette MQTT broker	Version 2.3.0 of the new Moquette MQTT broker component is available. New features Adds support for certificate chains.

Component	Details
Secret manager	Version 2.1.4 of the new secret manager is available. Bug fixes and improvements Fixes an issue where cached secrets were being removed when secret manager is deployed and Greengrass nucleus restarts.
Stream manager	Version 2.1.2 of the new stream manager is available. Bug fixes and improvements Fixes an issue on Windows OS that use a non-English language.

Release: AWS IoT Greengrass Core v2.8.1 software update on October 13, 2022

This release provides version 2.8.1 of the Greengrass nucleus component.

Release date: October 13, 2022

> **Note**
>
> If you are using Greengrass nucleus version 2.8.0, we strongly recommend that you upgrade to Greengrass nucleus version 2.8.1.

Release details

- Public component updates

Public component updates

The following table lists AWS-provided components that include new and updated features.

Important

When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.8.1 of the Greengrass nucleus is available. Bug fixes and improvements • Fixes an issue where deployment error codes were not generated correctly from Greengrass API errors. • Fixes an issue where fleet status updates send inaccurate information when a component reaches an ERR0RED state during a deployment. • Fixes an issue where deployments couldn't complete when Greengrass had more than 50 existing subscriptions.

Release: AWS IoT Greengrass Core v2.8.0 software update on October 7, 2022

This release provides version 2.8.0 of the Greengrass nucleus component and version 1.1.0 of the MQTT 5 broker (EMQX) component.

Release date: October 7, 2022

Release highlights

- **Deployment error codes** – The Greengrass nucleus now reports a deployment health status response that includes detailed error codes when a component deployment can't be completed. For more information, see Detailed deployment error codes.
- **Component error statuses** – The Greengrass nucleus now reports a component health status response that includes detailed error statuses when a component enters the BROKEN or ERRORED state. For more information, see Detailed component status codes.

Release details

- Public component updates

Public component updates

The following table lists AWS-provided components that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
>
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.8.0 of the Greengrass nucleus is available.

Component	Details
	New features • Updates the Greengrass nucleus to report a deployment health status response that includes detailed error codes when there is a problem deploying components to a core device. For more information, see Detailed deployment error codes. • Updates the Greengrass nucleus to report a component health status response that includes detailed error codes when a component enters the BROKEN or ERRORED state. For more information, see Detailed component status codes. • Expands status message fields to improve cloud availability informati on for devices. • Improves fleet status service robustness. Bug fixes and improvements • Allows a broken component to reinstall when its configuration changes. • Fixes an issue where a nucleus restart during bootstrap deployment causes a deployment to fail. • Fixes an issue in Windows where installation fails when a root path contains spaces. • Fixes an issue where a component shut down during a deployment uses the shutdown script of the new version. • Various shutdown improvements. • Additional minor fixes and improvements.
MQTT 5 broker (EMQX)	Version 1.1.0 of the MQTT 5 broker (EMQX) component is available. New features • Adds support for EMQX configurations including broker options and plug-ins. Bug fixes and improvements • Updates EMQX to version 4.4.9.

Release: AWS IoT Greengrass Core v2.7.0 software update on July 28, 2022

This release provides version 2.7.0 of the Greengrass nucleus component, version 2.1.0 of the stream manager component, and version 2.2.5 of the Lambda manager component.

Release date: July 28, 2022

Release highlights

- **Stream manager telemetry metrics** – Stream manager now automatically sends telemetry metrics to Amazon EventBridge, so you can create cloud applications that monitor and analyze the volume of data that your core devices upload. For more information, see Gather system health telemetry data from AWS IoT Greengrass core devices.
- **Custom certificate authority (CA)** – Client certificates signed by a custom certificate CA, where the CA isn't registered with AWS IoT, are now supported. For more information, see Use a device certificate signed by a private CA.

Release details

- Public component updates

Public component updates

The following table lists AWS-provided components that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
>
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you

create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.7.0 of the Greengrass nucleus is available. New features • Updates the Greengrass nucleus to send status updates to the AWS IoT Greengrass cloud when the core device applies a local deployment. • Adds support for client certificates signed by a custom certificate authority (CA), where the CA isn't registered with AWS IoT. To use this feature, you can set the new `greengrassDataPlaneEndpoint` configuration option to `iotdata`. For more information, see Use a device certificate signed by a private CA. Bug fixes and improvements • Fixes an issue where the Greengrass nucleus rolls back a deployment in certain scenarios when the nucleus is stopped or restarted. The nucleus now resumes the deployment after the nucleus restarts. • Updates the Greengrass installer to respect the `--start` argument when you specify to set up the software as a system service. • Updates the behavior of SubscribeToComponentUpdates to set the deployment ID in events where the nucleus updated a component. • Additional minor fixes and improvements.
Stream manager	Version 2.1.0 of the stream manager component is available. New features • Updates this component to automatically send telemetry metrics to Amazon EventBridge. For more information, see Gather system health telemetry data from AWS IoT Greengrass core devices. This feature requires v2.7.0 or later of the Greengrass nucleus component.

Component	Details
	• Version updated for Greengrass nucleus version 2.7.0 release.
Lambda manager	Version 2.2.5 of the Lambda manager component is available. New features • Adds support for MQTT topic wildcards in event sources where you subscribe to local publish/subscribe messages. This feature requires v2.6.0 or later of the Greengrass nucleus component. • Version updated for Greengrass nucleus version 2.7.0 release.

Release: AWS IoT Greengrass Core v2.6.0 software update on June 27, 2022

This release provides version 2.6.0 of the Greengrass nucleus component, new AWS-provided components, and updates to AWS-provided components.

Release date: June 27, 2022

Release highlights

- **Wildcards in local publish/subscribe topics** – You can now use MQTT wildcards when you subscribe to local publish/subscribe topics. For more information, see Publish/subscribe local messages and SubscribeToTopic.
- **Client device shadow support** – You can now interact with client device shadows in custom components and sync client device shadows with AWS IoT Core. For more information, see Interact with and sync client device shadows.
- **Local MQTT 5 support for client devices** – You can now deploy the EMQX MQTT 5 broker to use MQTT 5 features in communication between client devices and a core device. For more information, see MQTT 5 broker (EMQX) and Connect client devices to core devices.
- **Recipe variables in component configurations** – You can now use specific recipe variables in component configurations. You can use these recipe variables when you define a component's default configuration in a recipe or when you configure a component in a deployment. For more information, see Recipe variables and Use recipe variables in merge updates.

- **Wildcards in IPC authorization policies** – You can now use the * wildcard to match any combination of characters in interprocess communication (IPC) authorization policies. This wildcard enables you to allow access to multiple resources in a single authorization policy. For more information, see Wildcards in authorization policies.
- **IPC operations that manage local deployments and components** – You can now develop custom components that manage local deployments and view component details. For more information, see IPC: Manage local deployments and components.
- **IPC operations that authenticate and authorize client devices** – You can now use these operations to create a custom local broker component. For more information, see IPC: Authenticate and authorize client devices.

Release details

- Public component updates

Public component updates

The following table lists AWS-provided components that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
>
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.6.0 of the Greengrass nucleus is available.

Component	Details
	New features • Adds support for MQTT wildcards when you subscribe to local publish/subscribe topics. For more information, see Publish/subscribe local messages and SubscribeToTopic. • Adds support for recipe variables in component configurations, other than the *component_dependency_name* :configuration: *json_pointer* recipe variable. You can use these recipes variables when you define a component's DefaultConfiguration in a recipe or when you configure a component in a deployment. To enable this feature, set the interpolateComponentConfiguration configuration option to true. For more information, see Recipe variables and Use recipe variables in merge updates. • Adds full support for the * wildcard in interprocess communication (IPC) authorization policies. You can now specify the * character in a resource string to match any combination of characters. For more information, see Wildcards in authorization policies. • Adds support for custom components to call IPC operations that the Greengrass CLI uses. You can use these IPC operations to manage local deployments, view component details, and generate a password that you can use to sign in to the local debug console. For more information, see IPC: Manage local deployments and components. Bug fixes and improvements • Fixes an issue where dependent components wouldn't react when their hard dependencies restart or change states in certain scenarios. • Improves error messages that the core device reports to the AWS IoT Greengrass cloud service when a deployment fails. • Fixes an issue where the Greengrass nucleus applied a thing deployment twice in certain scenarios when the nucleus restarts. • Additional minor fixes and improvements. For more information, see the releases on GitHub.

Component	Details
MQTT 5 broker (EMQX)	Version 1.0.0 of the new EMQX MQTT 5 broker component is available. New features • Adds support for the local EMQX MQTT 5 broker. Client devices can connect to this MQTT broker to communicate with a core device using MQTT 5 features.
Shadow manager	Version 2.2.0 of the shadow manager component is available. New features • Adds support for the local shadow service over the local publish/subscribe interface. You can now communicate with the local publish/subscribe message broker on shadow MQTT topics to get, update, and delete shadows on the core device. This feature enables you to connect client devices to the local shadow service by using the MQTT bridge to relay messages on shadow topics between client devices and the local publish/subscribe interface. This feature requires v2.6.0 or later of the Greengrass nucleus component. To connect client devices to the local shadow service, you must also use v2.2.0 or later of the MQTT bridge component. • Adds the `direction` option that you can configure to customize the direction to sync shadows between the local shadow service and the AWS Cloud. You can configure this option to reduce bandwidth and connections to the AWS Cloud.

Component	Details
Client device auth	Version 2.2.0 of the client device auth component is available. New features • Adds support for custom components to call interprocess communication (IPC) operations to authenticate and authorize client devices. You can use these operations in a custom MQTT broker component, for example. For more information, see IPC: Authenticate and authorize client devices. • Adds the `maxActiveAuthTokens` , `cloudQueueSize` , and `threadPoolSize` options that you can configure to tune how this component performs.
MQTT bridge	Version 2.2.0 of the MQTT bridge component is available. New features • Adds support for MQTT topic wildcards (# and +) when you specify local publish/subscribe as the source message broker. This feature requires v2.6.0 or later of the Greengrass nucleus component. • Adds the `targetTopicPrefix` option, which you can specify to configure the MQTT bridge to add a prefix to the target topic when it relays a message.
Greengrass CLI	Version 2.6.0 of the Greengrass CLI is available. New features • Adds support for custom components to call interprocess communication (IPC) operations that the Greengrass CLI uses. You can use these IPC operations to manage local deployments, view component details, and generate a password that you can use to sign in to the local debug console. For more information, see IPC: Manage local deployments and components. Bug fixes and improvements • Additional minor fixes and improvements.

Release: AWS IoT Greengrass Core v2.5.6 software update on May 31, 2022

This release provides version 2.5.6 of the Greengrass nucleus component and version 2.2.4 of the log manager component.

Release date: May 31, 2022

Release details

- Public component updates

Public component updates

The following table lists AWS-provided components that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
>
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.5.6 of the Greengrass nucleus is available. New features • Adds support for hardware security modules that use ECC keys. You can use a hardware security module (HSM) to securely store

Component	Details
	the device's private key and certificate. For more information, see Hardware security integration. Bug fixes and improvements • Fixes an issue where the deployment never completes when you deploy a component with a broken install script in certain scenarios. • Improves performance during startup. • Additional minor fixes and improvements.
Log manager	Version 2.2.4 of the log manager component is available. Bug fixes and improvements • Improves stability when handling invalid configurations. • Additional minor fixes and improvements.

Release: AWS IoT Greengrass Core v2.5.5 software update on April 6, 2022

This release provides version 2.5.5 of the Greengrass nucleus component.

Release date: April 6, 2022

Release details

- Public component updates

Public component updates

The following table lists AWS-provided components that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those

devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.5.5 of the Greengrass nucleus is available. New features • Adds the `GG_ROOT_CA_PATH` environment variable for components, so you can access the root certificate authority (CA) certificate in custom components. Bug fixes and improvements • Adds support for Windows devices that use a display language other than English. • Updates how the Greengrass nucleus parses Boolean installer arguments, so you can specify a Boolean argument without a Boolean value to specify a `true` value. For example, you can now specify `--provision` instead of `--provision true` to install with automatic resource provisioning. • Fixes an issue where the core device didn't report its status to the AWS IoT Greengrass cloud service after provisioning in certain scenarios. • Additional minor fixes and improvements.

Release: AWS IoT Greengrass Core v2.5.4 software update on March 23, 2022

This release provides version 2.5.4 of the Greengrass nucleus component and version 2.0.10 of the Lambda launcher component.

Release date: March 23, 2022

Release details

- Public component updates

Public component updates

The following table lists AWS-provided components that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.5.4 of the Greengrass nucleus is available. Bug fixes and improvements • General bug fixes and improvements.
Lambda launcher	Version 2.0.10 of the Lambda launcher component is available. Bug fixes and improvements • General bug fixes and improvements.

Release: AWS IoT Greengrass Core v2.5.3 software update on January 6, 2022

This release provides version 2.5.3 of the Greengrass nucleus component and the new PKCS#11 provider component.

Release date: January 6, 2022

Release highlights

- **Hardware security integration**—You can now configure the AWS IoT Greengrass Core software to use a private key and certificate that you securely store in a hardware security module (HSM). For more information, see Hardware security integration.

Release details

- Public component updates

Public component updates

The following table lists AWS-provided components that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
>
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.5.3 of the Greengrass nucleus is available. New features • Adds support for hardware security integration. You can use a hardware security module (HSM) to securely store the device's private key and certificate. For more information, see Hardware security integration. Bug fixes and improvements • Fixes an issue with runtime exceptions while the nucleus establishes MQTT connections with AWS IoT Core.
PKCS#11 provider	Version 2.0.0 of the PKCS#11 provider component is available. New features • Adds support for hardware security integration. You can use a hardware security module (HSM) to securely store the device's private key and certificate. For more information, see Hardware security integration.

Release: AWS IoT Greengrass Core v2.5.2 software update on December 3, 2021

This release provides version 2.5.2 of the Greengrass nucleus component.

Release date: December 3, 2021

Release details

- Public component updates

Public component updates

The following table lists AWS-provided components that include new and updated features.

Important

When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.5.2 of the Greengrass nucleus is available. Bug fixes and improvements • Fixes an issue where after the Greengrass nucleus updates, the Windows service fails to start again after you stop it or reboot the device.
AWS IoT Device Defender	Version 3.0.1 of the AWS IoT Device Defender component is available. This version of the AWS IoT Device Defender component expects different configuration parameters than version 2.x. If you use a non-default configuration for version 2.x, and you want to upgrade from v2.x to v3.x, you must update the component's configuration. For more information, see AWS IoT Device Defender component configuration. New features • Adds support for core devices that run Windows. • Changes the component type from Lambda component to generic component. This component now no longer depends on the legacy subscription router component to create subscriptions.

Component	Details
	• Adds the new `UseInstaller` configuration parameter that lets you optionally disable the installation script that installs component dependencies.

Release: AWS IoT Greengrass Core v2.5.1 software update on November 23, 2021

This release provides version 2.5.1 of the Greengrass nucleus component.

Release date: November 23, 2021

Release details

- Public component updates

Public component updates

The following table lists AWS-provided components that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.5.1 of the Greengrass nucleus is available. Bug fixes and improvements • Adds support for 32-bit versions of the Java Runtime Environment (JRE) on Windows. • Changes thing group removal behavior for core devices whose AWS IoT policy doesn't grant the `greengrass:ListThingGroupsForCoreDevice` permission. With this version, the deployment continues, logs a warning, and doesn't remove components when you remove the core device from a thing group. For more information, see Deploy AWS IoT Greengrass components to devices. • Fixes an issue with system environment variables that the Greengrass nucleus makes available to Greengrass component processes. You can now restart a component for it to use the latest system environment variables.

Release: AWS IoT Greengrass Core v2.5.0 software update on November 12, 2021

This release provides version 2.5.0 of the Greengrass nucleus component, new AWS-provided components, and updates to AWS-provided components.

Release date: November 12, 2021

Release highlights

- **Windows device support**—You can now run the AWS IoT Greengrass Core software on devices running Windows operating systems. For more information, see Greengrass feature compatibility.
- **New thing group removal behavior**—You can now remove a core device from a thing group to remove that thing group's components in the next deployment to that device.

> **Important**
>
> As a result of this change, a core device's AWS IoT policy must have the `greengrass:ListThingGroupsForCoreDevice` permission. If you used the AWS IoT Greengrass Core software installer to provision resources, the default AWS IoT policy allows `greengrass:*`, which includes this permission. For more information, see Device authentication and authorization for AWS IoT Greengrass.

- **Hardware security support**—You can now configure the AWS IoT Greengrass Core software to use a hardware security module (HSM), so you can securely store the device's private key and certificate. For more information, see Hardware security integration.
- **HTTPS proxy support**—You can now configure the AWS IoT Greengrass Core software to connect through HTTPS proxies. For more information, see Connect on port 443 or through a network proxy.

Release details

- Platform support updates
- Public component updates

Platform support updates

Platform	Details
Windows	AWS IoT Greengrass now supports running the AWS IoT Greengrass Core software on the following versions of Windows: • Windows 10 • Windows Server 2019 For more information, see Greengrass feature compatibility.

Public component updates

The following table lists AWS-provided components that include new and updated features.

Important

When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.5.0 of the Greengrass nucleus is available. New features • Adds support for core devices that run Windows. • Change the behavior of thing group removal. With this version, you can remove a core device from a thing group to uninstall that thing group's components in the next deployment. As a result of this change, a core device's AWS IoT policy must have the `greengrass:ListThingGroupsForCoreDevice` permission. If you used the AWS IoT Greengrass Core software installer to provision resources, the default AWS IoT policy allows `greengrass:*` , which includes this permission. For more information, see Device authentication and authorization for AWS IoT Greengrass. • Adds support for HTTPS proxy configurations. For more information, see Connect on port 443 or through a network proxy. • Adds the new `windowsUser` configuration parameter. You can use this parameter to specify the default user to use to run components on a Windows core device. For more information, see Configure the user that runs components.

Component	Details
	• Adds the new `httpClient` configuration options that you can use to customize HTTP request timeouts to improve performance on slow networks. For more information, see the httpClient configuration parameter. Bug fixes and improvements • Fixes the bootstrap lifecycle option to restart the core device from a component. • Adds support for hyphens in recipe variables. • Fixes IPC authorization for on-demand Lambda function components. • Improves log messages and changes non-critical logs from `INFO` to `DEBUG` level, so logs are more useful. • Removes the `iot:DescribeCertificate` permission from the default token exchange role that the Greengrass nucleus creates when you install the AWS IoT Greengrass Core software with automatic provisioning. This permission isn't used by the Greengrass nucleus. • Fixes an issue so that the automatic provisioning script doesn't require the `iam:GetPolicy` permission if `iam:CreatePolicy` is available for the same policy. • Additional minor fixes and improvements.
Greengrass CLI	Version 2.5.0 of the Greengrass CLI is available. New features • Adds support for core devices that run Windows. • Adds the new `AuthorizedWindowsGroups` configuration parameter that you can specify to authorize system groups to use the Greengrass CLI on Windows devices. • Adds the `windowsUser` parameter for local deployments. You can use this parameter specify the user to use to run components on a Windows core device.

Component	Details
CloudWatch metrics	Version 3.0.0 of the CloudWatch metrics component is available. This version of the CloudWatch metrics component expects different configuration parameters than version 2.x. If you use a non-default configuration for version 2.x, and you want to upgrade from v2.x to v3.x, you must update the component's configuration. For more information, see CloudWatch metrics component configuration. New features • Adds support for core devices that run Windows. • Changes the component type from Lambda component to generic component. This component now no longer depends on the legacy subscription router component to create subscriptions. • Adds new `InputTopic` configuration parameter to specify the topic to which the component subscribes to receive messages. • Adds new `OutputTopic` configuration parameter to specify the topic to which the component publishes status responses. • Adds new `PubSubToIoTCore` configuration parameter to specify whether to publish and subscribe to AWS IoT Core MQTT topics. • Adds the new `UseInstaller` configuration parameter that lets you optionally disable the installation script that installs component dependencies. Bug fixes and improvements Adds support for duplicate timestamps in input data.

Component	Details
Lambda manager	Version 2.2.0 of the Lambda manager component is available. Bug fixes and improvements • Fixes an issue where Lambda functions couldn't write logs after a restart. • Fixes an issue where the legacy subscription router sends duplicate messages when there are wildcards in the topic. • Fixes an issue where non-pinned Lambda functions couldn't use the Greengrass interprocess communication (IPC) library in the AWS IoT Device SDK.

Release: AWS IoT Greengrass Core v2.4.0 software update on August 3, 2021

This release provides version 2.4.0 of the Greengrass nucleus component, new AWS-provided components, and updates to AWS-provided components.

Release date: August 3, 2021

Release highlights

- **System resource limits**—The Greengrass nucleus component now supports system resource limits. You can configure the maximum amount of CPU and RAM usage that each component's processes can use on the core device. For more information, see Configure system resource limits for components.
- **Pause/resume components**—The Greengrass nucleus now supports pausing and resuming components. You can use the interprocess communication (IPC) library to develop custom components that pause and resume other components' processes. For more information, see PauseComponent and ResumeComponent.
- **Install with AWS IoT fleet provisioning**—Use the new AWS IoT fleet provisioning plugin to install the AWS IoT Greengrass Core software on devices that connect to AWS IoT to provision required AWS resources. Devices use a claim certificate to provision. You can embed the claim certificate on devices during manufacturing, so each device can provision as soon as it comes

online. For more information, see Install AWS IoT Greengrass Core software with AWS IoT fleet provisioning.

- **Install with custom provisioning**—Develop a custom provisioning plugin to provision required AWS resources when you install the AWS IoT Greengrass Core software on devices. You can create a Java application that runs during installation to set up Greengrass core devices for your custom use case. For more information, see Install AWS IoT Greengrass Core software with custom resource provisioning.

Release details

- Public component updates

Public component updates

The following table lists AWS-provided components that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
>
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.4.0 of the Greengrass nucleus is available. New features • Adds support for system resource limits. You can configure the maximum amount of CPU and RAM usage that each component

Component	Details
	's processes can use on the core device. For more information, see Configure system resource limits for components. • Adds IPC operations to pause and resume components. For more information, see PauseComponent and ResumeComponent. • Adds support for provisioning plugins. You can specify a JAR file to run during installation to provision required AWS resources for a Greengrass core device. The Greengrass nucleus includes an interface that you can implement to develop custom provisioning plugins. For more information, see Install AWS IoT Greengrass Core software with custom resource provisioning. • Adds the optional `thing-name-policy` argument to the AWS IoT Greengrass Core software installer. You can use this option to specify an existing or custom AWS IoT policy when you install the AWS IoT Greengrass Core software with automatic resource provisioning. Bug fixes and improvements • Updates logging configuration on startup. This fixes an issue where the logging configuration wasn't applied on startup. • Updates the nucleus loader symlink to point to the component store in the Greengrass root folder during installation. This update enables you to delete the JAR file and other nucleus artifacts that you download when you install the AWS IoT Greengrass Core software. • Additional minor fixes and improvements. For more information, see the releases on GitHub.
Greengrass CLI	Version 2.4.0 of the Greengrass CLI is available. New features • Adds support for system resource limits. When you create a local deployment, you can configure the maximum amount of CPU and RAM usage that each component's processes can use on the core device. For more information, see Configure system resource limits for components and the deployment create command.

Component	Details
AWS IoT fleet provisioning by claim	The AWS IoT fleet provisioning by claim plugin is now available. For more information, see Install AWS IoT Greengrass Core software with AWS IoT fleet provisioning. New features • Adds support to install the AWS IoT Greengrass Core software with AWS IoT fleet provisioning. During installation, devices connect to AWS IoT to provision required AWS resources and download device certifica tes to use for regular operations.

Release: AWS IoT Greengrass Core v2.3.0 software update on June 29, 2021

This release provides version 2.3.0 of the Greengrass nucleus component.

Release date: June 29, 2021

Release highlights

- **Large configuration support**—The Greengrass nucleus component now supports deployment documents up to 10 MB. You can now deploy larger configuration updates to Greengrass components.

> **Note**
>
> To use this feature, a core device's AWS IoT policy must allow the `greengrass:GetDeploymentConfiguration` permission. If you used the AWS IoT Greengrass Core software installer to provision resources, your core device's AWS IoT policy allows `greengrass:*`, which includes this permission. For more information, see Device authentication and authorization for AWS IoT Greengrass.

Release details

- Public component updates

Public component updates

The following table lists AWS-provided components that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
>
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.3.0 of the Greengrass nucleus is available. New features • Adds support for deployment configuration documents up to 10 MB, up from 7 KB (for deployments that target things) or 31 KB (for deployments that target thing groups). To use this feature, a core device's AWS IoT policy must allow the `greengrass:GetDeploymentConfiguration` permission. If you used the AWS IoT Greengrass Core software installer to provision resources, your core device's AWS IoT policy allows `greengrass:*` , which includes this permission. For more information, see Device authentication and authorization for AWS IoT Greengrass. • Adds the `iot:thingName` recipe variable. You can use this recipe variable to get the name of the core device's AWS IoT thing in a recipe. For more information, see Recipe variables.

Component	Details
	Bug fixes and improvements • Additional minor fixes and improvements. For more information, see the releases on GitHub.

Release: AWS IoT Greengrass Core v2.2.0 software update on June 18, 2021

This release provides version 2.2.0 of the Greengrass nucleus component, new AWS-provided components, and updates to AWS-provided components.

Release date: June 18, 2021

Release highlights

- **Client device support**—The new AWS-provided client device components enable you to connect client devices to your core devices using cloud discovery. You can sync client devices with AWS IoT Core and interact with client devices in Greengrass components. For more information, see Interact with local IoT devices.
- **Local shadow service**—The new shadow manager component enables the local shadow service on your core devices. You can use this shadow service to interact with local shadows while offline using the Greengrass interprocess communication (IPC) libraries in the AWS IoT Device SDK. You can also use the shadow manager component to synchronize local shadow states with AWS IoT Core. For more information, see Interact with device shadows.

Release details

- Public component updates

Public component updates

The following table lists AWS-provided components that include new and updated features.

Important

When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.2.0 of the Greengrass nucleus is available. New features • Adds IPC operations for local shadow management. Bug fixes and improvements • Reduces the size of the JAR file. • Reduces memory usage. • Fixes issues where the log configuration wasn't updated in certain cases. • Additional minor fixes and improvements. For more information, see the releases on GitHub.
Shadow manager	Version 2.0.0 of the new shadow manager component is available. New features • Adds support for classic and named shadows. • Adds support for local shadow management using IPC. • Adds support for shadow synchronization with AWS IoT Core.

Component	Details
Client device auth	Version 2.0.0 of the new client device auth component is available. New features • Adds support for Greengrass client devices, which are local IoT devices that connect to a core device over MQTT. • Adds support for authentication and authorization of client devices and their MQTT actions.
Moquette MQTT broker	Version 2.0.0 of the new Moquette MQTT broker component is available. New features • Adds support for a local Moquette MQTT broker that handles communication with client devices.
MQTT bridge	Version 2.0.0 of the new MQTT bridge component is available. New features • Adds support to relay messages between the local MQTT broker, the local Greengrass publish/subscribe broker, and the AWS IoT Core MQTT broker.
IP detector	Version 2.0.0 of the new IP detector component is available. New features • Adds support to report a core device's local MQTT broker endpoints to the AWS IoT Greengrass cloud service for client devices to connect.
Log manager	Version 2.1.1 of the log manager component is available. Bug fixes and improvements • Fixes an issue where the system log configuration wasn't updated in certain cases.

Component	Details
DLR object detection	Version 2.1.2 of the DLR object detection is available. Bug fixes and improvements • Fixes an image scaling issue that resulted in inaccurate bounding boxes in the sample DLR object detection inference results.
TensorFlow Lite object detection	Version 2.1.1 of the TensorFlow Lite object detection is available. Bug fixes and improvements • Fixes an image scaling issue that resulted in inaccurate bounding boxes in the sample TensorFlow Lite object detection inference results.

Release: AWS IoT Greengrass Core v2.1.0 software update on April 26, 2021

This release provides version 2.1.0 of the Greengrass nucleus component and updates AWS-provided components.

Release date: April 26, 2021

Release highlights

- **Docker Hub and Amazon Elastic Container Registry (Amazon ECR) integration**—The new Docker application manager component enables you to download public or private images from Amazon ECR. You can also use this component to download public images from Docker Hub and AWS Marketplace. For more information, see Run a Docker container.
- **Dockerfile and Docker images for AWS IoT Greengrass Core software**—You can use the Greengrass Docker image to run AWS IoT Greengrass in a Docker container that uses Amazon Linux 2 as the base operating system. You can also use the AWS IoT Greengrass Dockerfile to build your own Greengrass image. For more information, see Run AWS IoT Greengrass Core software in a Docker container.
- **Support for additional machine learning frameworks and platforms**—You can deploy sample machine learning inference components that use pre-trained models to perform sample image classification and object detection using TensorFlow Lite 2.5.0 and DLR 1.6.0. This release also

extends sample machine learning support for Armv8 (AArch64) devices. For more information, see Perform machine learning inference.

Release details

- Platform support updates
- Public component updates

Platform support updates

Platform	Details
Docker	A Dockerfile and Docker image for AWS IoT Greengrass are now available. Dockerfile AWS IoT Greengrass provides a Dockerfile to build a container image that has AWS IoT Greengrass Core software and dependencies installed on an Amazon Linux 2 (x86_64) base image. You can modify the base image in the Dockerfile to run AWS IoT Greengrass on a different platform architecture. Docker image AWS IoT Greengrass provides a pre-built Docker image that has AWS IoT Greengrass Core software and dependencies installed on an Amazon Linux 2 (x86_64) base image. For more information, see Run AWS IoT Greengrass Core software in a Docker container.

Public component updates

The following table lists AWS-provided components that include new and updated features.

Important

When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details

Greengrass nucleus	Version 2.1.0 of the Greengrass nucleus is available.
	
	New features
	• Supports downloading Docker images from private repositories in Amazon ECR.
	• Adds the following parameters to customize the MQTT configuration on core devices:
	  • `maxInFlightPublishes` – The maximum number of unacknowledged MQTT QoS 1 messages that can be in flight at the same time.
	  • `maxPublishRetry` – The maximum number of times to retry a message that fails to publish.
	• Adds the `fleetstatusservice` configuration parameter to configure the interval at which the core device publishes device status to the AWS Cloud.
	• Additional minor fixes and improvements. For more information, see the releases on GitHub.
	
	Bug fixes and improvements
	• Fixes an issue that caused shadow deployments to be duplicated when the nucleus restarts.

Component	Details
	• Fixes an issue that caused the nucleus to crash when it encountered a service load exception. • Improves component dependency resolution to fail a deployment that includes a circular dependency. • Fixes an issue that prevented a plugin component from being redeployed if that component had been previously removed from the core device. • Fix an issue that caused the `HOME` environment variable to be set to the *`/greengrass/v2`* `/work` directory for Lambda components or for components that run as root. The `HOME` variable is now correctly set to the home directory for the user that runs the component. • Additional minor fixes and improvements. For more information, see the releases on GitHub.
Docker application manager	Version 2.0.0 of the new Docker application manager component is available . New features • Manages credentials to download images from private repositories in Amazon ECR. • Downloads public images from Amazon ECR, Docker Hub, and AWS Marketplace.
Lambda launcher	Version 2.0.4 of the Lambda launcher component is available. Bug fixes and improvements • Fixes an issue where the component doesn't correctly pass `AddGroupOwner` to the Lambda function container.

Component	Details
Legacy subscription router	Version 2.1.0 of the legacy subscription router component is available. Bug fixes and improvements • Adds support to specify component names instead of ARNs for `source` and `target`. If you specify a component name for a subscription, you don't need to reconfigure the subscription each time the version of the Lambda function changes.
Local debug console	Version 2.1.0 of the local debug console component is available. New features • Uses HTTPS to secure your connection to the local debug console. HTTPS is enabled by default. Bug fixes and improvements • You can dismiss flashbar messages in the configuration editor.
Log manager	Version 2.1.0 of the log manager component is available. Bug fixes and improvements • Use defaults for `logFileDirectoryPath` and `logFileRegex` that work for Greengrass components that print to standard output (stdout) and standard error (stderr). • Correctly route traffic through a configured network proxy when uploading logs to CloudWatch Logs. • Correctly handle colon characters (`:`) in log stream names. CloudWatch Logs log stream names don't support colons. • Simplify log stream names by removing thing group names from the log stream. • Remove an error log message that prints during normal behavior.

Component	Details
DLR image classification	Version 2.1.1 of the DLR image classification component is available. New features • Use Deep Learning Runtime v1.6.0. • Add support for sample image classification on Armv8 (AArch64) platforms. This extends machine learning support for Greengrass core devices running NVIDIA Jetson, such as the Jetson Nano. • Enable camera integration for sample inference. Use the new `UseCamera` configuration parameter to enable the sample inference code to access the camera on your Greengrass core device and run inference locally on the captured image. • Add support for publishing inference results to the AWS Cloud. Use the new `PublishResultsOnTopic` configuration parameter to specify the topic on which you want to publish results. • Add the new `ImageDirectory` configuration parameter that enables you to specify a custom directory for the image on which you want to perform inference. Bug fixes and improvements • Write inference results to the component log file instead of a separate inference file. • Use the AWS IoT Greengrass Core software logging module to log component output. • Use the AWS IoT Device SDK to read the component configuration and apply configuration changes.

Component	Details
DLR object detection	Version 2.1.1 of the DLR object detection component is available. New features • Use Deep Learning Runtime v1.6.0. • Add support for sample object detection on Armv8 (AArch64) platforms. This extends machine learning support for Greengrass core devices running NVIDIA Jetson, such as the Jetson Nano. • Enable camera integration for sample inference. Use the new `UseCamera` configuration parameter to enable the sample inference code to access the camera on your Greengrass core device and run inference locally on the captured image. • Add support for publishing inference results to the AWS Cloud. Use the new `PublishResultsOnTopic` configuration parameter to specify the topic on which you want to publish results. • Add the new `ImageDirectory` configuration parameter that enables you to specify a custom directory for the image on which you want to perform inference. Bug fixes and improvements • Write inference results to the component log file instead of a separate inference file. • Use the AWS IoT Greengrass Core software logging module to log component output. • Use the AWS IoT Device SDK to read the component configuration and apply configuration changes.
DLR image classification model store	Version 2.1.1 of the DLR image classification model store component is available. New features • Add a sample ResNet-50 image classification model for Armv8 (AArch64) platforms. This extends machine learning support for Greengrass core devices running NVIDIA Jetson, such as the Jetson Nano.

Component	Details
DLR object detection model store	Version 2.1.1 of the DLR object detection model store component is available. New features • Add a sample YOLOv3 object detection model for Armv8 (AArch64) platforms. This extends machine learning support for Greengrass core devices running NVIDIA Jetson, such as the Jetson Nano.
DLR installer	Version 1.6.1 of the DLR component is available. New features • Install Deep Learning Runtime v1.6.0 and its dependencies. • Add support for installing DLR on Armv8 (AArch64) platforms. This extends machine learning support for Greengrass core devices running NVIDIA Jetson, such as the Jetson Nano. Bug fixes and improvements • Install the AWS IoT Device SDK in the virtual environment to read the component configuration and apply configuration changes. • Additional minor bug fixes and improvements.
TensorFlow Lite image classification	Version 2.1.0 of the new TensorFlow Lite image classification component is available. New features • Add support for sample image classification inference using TensorFlow Lite.
TensorFlow Lite object detection	Version 2.1.0 of the new TensorFlow Lite object detection component is available. New features • Add support for sample object detection inference using TensorFlow Lite.

Component	Details
TensorFlow Lite image classification model store	Version 2.1.0 of the new TensorFlow Lite image classification model store component is available. New features • Provide a pre-trained MobileNet v1 quantized model for sample image classification inference using TensorFlow Lite.
TensorFlow Lite object detection model store	Version 2.1.0 of the new TensorFlow Lite object detection model store component is available. New features • Provide a pre-trained Single Shot Detection (SSD) MobileNet model trained on the COCO dataset for sample object detection inference using TensorFlow Lite.
TensorFlow Lite	Version 2.5.0 of the new TensorFlow Lite component is available. New features • Install TensorFlow Lite v1.6.0 and its dependencies in a virtual environment on Armv7, Armv8 (AArch64), and x86_64 platforms.

Release: AWS IoT Greengrass Core v2.0.5 software update on March 09, 2021

This release provides version 2.0.5 of the Greengrass nucleus component and updates AWS-provided components. It fixes an issue with network proxy support and an issue with the Greengrass data plane endpoint in AWS China Regions.

Release date: March 09, 2021

Public component updates

The following table lists AWS-provided components that include new and updated features.

Important

When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.0.5 of the Greengrass nucleus is available. Bug fixes and improvements • Correctly routes traffic through a configured network proxy when downloading AWS-provided components. • Use the correct Greengrass data plane endpoint in AWS China Regions.

Release: AWS IoT Greengrass Core v2.0.4 software update on February 04, 2021

This release provides version 2.0.4 of the Greengrass nucleus component. It includes the new `greengrassDataPlanePort` parameter to configure HTTPS communication over port 443 and fixes bugs. The minimal IAM policy now requires the `iam:GetPolicy` and `sts:GetCallerIdentity` when the AWS IoT Greengrass Core software installer is run with `--provision true`.

Release date: February 04, 2021

Public component updates

The following table lists AWS-provided components that include new and updated features.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
>
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Component	Details
Greengrass nucleus	Version 2.0.4 of the Greengrass nucleus is available. New features • Enables HTTPS traffic over port 443. You can use the new `greengrasDataPlanePort` configuration parameter for version 2.0.4 of the nucleus component to configure HTTPS communication to travel over port 443 instead of the default port 8443. For more information, see Configure HTTPS over port 443. • Adds the work path recipe variable. You can use this recipe variable to get the path to components' work folders, which you can use to share files between components and their dependencies. For more informati on, see the work path recipe variable. Bug fixes and improvements • Prevents the creation of the token exchange AWS Identity and Access Management (IAM) role policy if a role policy already exists.

Component	Details
	As a result of this change, the installer now requires the `iam:GetPolicy` and `sts:GetCallerIdentity` when run with `--provision true`. For more information, see Minimal IAM policy for installer to provision resources. • Correctly handles the cancellation of a deployment that has not yet been registered successfully. • Updates the configuration to remove older entries with newer timestamps when rolling back a deployment. • Additional minor fixes and improvements. For more information, see the releases on GitHub.

Migrate from AWS IoT Greengrass Version 1

AWS IoT Greengrass Version 2 is a major version release of the AWS IoT Greengrass Core software, APIs, and console. AWS IoT Greengrass V2 introduces several improvements to AWS IoT Greengrass V1, such as modular applications, deployments to large fleets of devices, and support for additional platforms.

> **Note**
>
> End of support notice: On October 7, 2026, AWS ended support for AWS IoT Greengrass Version 1. The AWS IoT Greengrass V1 API operations and resources are no longer available.

Follow instructions in this guide to migrate from AWS IoT Greengrass V1 to AWS IoT Greengrass V2.

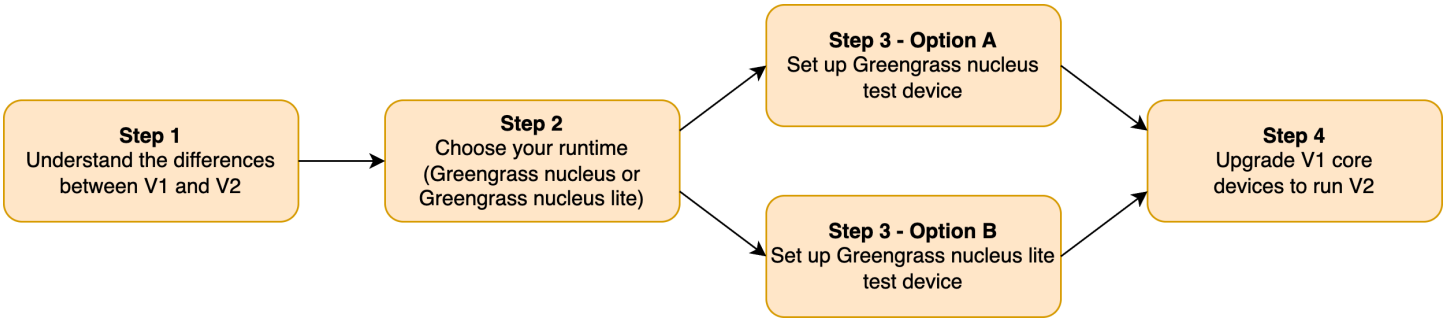
Migration overview

At a high level, you can use the following procedure to upgrade core devices from AWS IoT Greengrass V1 to AWS IoT Greengrass V2.

Before migrating, you'll choose between two runtime options:

- **Greengrass nucleus** (lower migration effort, full feature support)
- **Greengrass nucleus lite** (higher migration effort, designed for resource-constrained devices).

The exact procedure that you follow depends on your device resources, required features, and specific environment requirements.



1. **Understand the differences between V1 and V2**

   AWS IoT Greengrass V2 introduces new fundamental concepts for device fleets and deployable software, and V2 simplifies several concepts from V1.

   The AWS IoT Greengrass V2 cloud service and AWS IoT Greengrass Core software v2.x aren't backward compatible with the AWS IoT Greengrass V1 cloud service and AWS IoT Greengrass Core software v1.x. As a result, AWS IoT Greengrass V1 over-the-air (OTA) updates can't upgrade core devices from V1 to V2.

2. **Choose your runtime (Greengrass nucleus or Greengrass nucleus lite)**

   Decide between Greengrass nucleus or Greengrass nucleus lite based on your device resources and feature requirements:

   - **Greengrass nucleus path**: Lower migration effort. Lambda functions can be imported as Lambda components with minimal code changes. Supports V1 features (local shadow service, client devices, connectors).
   - **Greengrass nucleus lite path**: Higher migration effort. Lambda functions need to be converted to generic components, requiring code changes to use AWS IoT Device SDK V2/ AWS IoT Greengrass Component SDK instead of AWS IoT Greengrass Core SDK. Does not support local shadow service, client devices, or connectors.

3. **Set up a new device to test V1 applications on V2**

   To minimize risk to your devices in production, create a new device to test your V1 applications on V2. Choose the setup guide based on your runtime selection:

   - **Option A - Greengrass nucleus runtime**: Set up a new device to test V1 applications on V2. Import Lambda functions as Lambda components with minimal code changes.
   - **Option B - Greengrass nucleus lite runtime**: Set up a new device to test V1 applications on V2 (Greengrass nucleus lite). Convert Lambda functions to generic components using AWS IoT Device SDK.

4. **Upgrade V1 core devices to run V2**

   After testing on a new device, upgrade your existing V1 core devices to run the AWS IoT Greengrass Core software v2.x and AWS IoT Greengrass V2 components. To migrate a fleet of devices from V1 to V2, you repeat this step for each device in the fleet.

Differences between AWS IoT Greengrass V1 and AWS IoT Greengrass V2

AWS IoT Greengrass V2 introduces new fundamental concepts for devices, fleets, and deployable software. This section describes the V1 concepts that are different in V2.

Greengrass concepts and terminology

Concept	AWS IoT Greengrass V1	AWS IoT Greengrass V2
Application code	In AWS IoT Greengrass V1, Lambda functions define the software that runs on core devices. In each Greengrass group, you define subscript ions and local resources that the function uses. For Lambda functions that the AWS IoT Greengrass Core software runs in a containerized Lambda runtime environme nt, you define container parameters, such as memory limits.	In AWS IoT Greengrass V2, *components* are the software modules that run on core devices. • Each component has a *recipe* that defines the component's metadata, parameters, dependenc ies, and scripts to run at each step in the component lifecycle. • The recipe also defines the component's *artifacts*, which are binary files, such as scripts, compiled code, and static resources. • When you deploy a component to a core device, the core device downloads the component recipe and artifacts to run the component. You can import your V1 Lambda functions as components that run in a

Concept	AWS IoT Greengrass V1	AWS IoT Greengrass V2
		Lambda runtime environment in AWS IoT Greengrass V2. When you import the Lambda function, you specify the subscriptions, local resources, and container parameters for the function. For more information, see Step 2: Create and deploy AWS IoT Greengrass V2 components to migrate AWS IoT Greengrass V1 applications. For more information about how to create custom components, see Develop AWS IoT Greengrass components.

Concept	AWS IoT Greengrass V1	AWS IoT Greengrass V2
AWS IoT Greengrass groups and deployments	In AWS IoT Greengrass V1, a group defines the core device, the settings and software for that core device, and the list of AWS IoT things that can connect to that core device. You create a deployment to send a group's configuration to a core device.	In AWS IoT Greengrass V2, you use *deployments* to define the software components and configurations that run on core devices. • Each deployment targets a single core device (which is an AWS IoT thing) or an AWS IoT thing group that can contain multiple core devices. • Deployments to thing groups are continuous, so when you add a core device to a thing group, it receives the software configuration for that group. For more information, see Deploy AWS IoT Greengrass components to devices. In AWS IoT Greengrass V2, you can also create local deployments using the Greengrass CLI to test custom software components on the device where you develop them. For more information, see Create AWS IoT Greengrass components.

Concept	AWS IoT Greengrass V1	AWS IoT Greengrass V2
AWS IoT Greengrass Core software	In AWS IoT Greengrass V1, the AWS IoT Greengrass Core software is a single package that contains the software and all of its features. The edge device on which you install the AWS IoT Greengrass Core software is called a Greengrass core.	In AWS IoT Greengrass V2, the AWS IoT Greengrass Core software is modular, so that you can choose what to install to control the memory footprint. • The Greengrass nucleus component is the minimum required installation of the AWS IoT Greengrass Core software. The edge device on which you install the nucleus is called a Greengrass core device. • The nucleus handles deployments, orchestration, and lifecycle management of other components on the core device. • Features such as stream manager, secret manager, and log manager are components that you deploy only when you need those features. For more information, see AWS-provided components.

Concept	AWS IoT Greengrass V1	AWS IoT Greengrass V2
Connectors	In AWS IoT Greengrass V1, connectors are prebuilt modules that you deploy to AWS IoT Greengrass V1 core devices to interact with local infrastructure, device protocols, AWS, and other cloud services.	In AWS IoT Greengrass V2, AWS provides Greengrass components that implement the functionality provided by connectors in V1. The following AWS IoT Greengrass V2 components provide Greengrass V1 connector functionality: • CloudWatch metrics component • AWS IoT Device Defender component • Firehose component • Modbus-RTU protocol adapter component • Amazon SNS component For more information, see AWS-provided components.

Concept	AWS IoT Greengrass V1	AWS IoT Greengrass V2
Connected devices (Greengrass devices)	In AWS IoT Greengrass V1, connected devices are AWS IoT things that you add to a Greengrass group to connect to the core device in that group and communicate over MQTT. You must deploy that group each time that you add or remove a connected device. You use subscriptions to relay messages between connected devices, AWS IoT Core, and applications on the core device.	In AWS IoT Greengrass V2, connected devices are called Greengrass client devices. • You associate client devices to core devices to connect them and communicate over MQTT. • To authorize client devices to connect, you define authorization policies that can apply to groups of client devices, so you don't need to create a deployment to add or remove a client device. • To relay messages between client devices, AWS IoT Core, and Greengrass components, you can configure an optional MQTT bridge component. In both AWS IoT Greengrass V1 and AWS IoT Greengrass V2, devices can run FreeRTOS or use the AWS IoT Device SDK or Greengrass discovery API to get information about core devices to which they can connect. The Greengrass discovery API is backward compatible, so if you have client devices that connect

Concept	AWS IoT Greengrass V1	AWS IoT Greengrass V2
		to a V1 core device, you can connect them to a V2 core device without changing their code. For more information about client devices, see Interact with local IoT devices.
Local resources	In AWS IoT Greengrass V1, Lambda functions that run in containers can be configure d to access volumes and devices on the core device's file system. These file system resources are known as local resources.	In AWS IoT Greengrass V2, you can run components that are Lambda functions, Docker containers, or native operating system processes or custom runtimes. • When you import a containerized Lambda function as a component, you must specify the local resources that the function uses. • Non-containerized Lambda functions and non-Lambd a components can work directly with local resources on core devices, so you don't need to specify the local resources that the component uses.

Concept	AWS IoT Greengrass V1	AWS IoT Greengrass V2
Local shadow service	In AWS IoT Greengrass V1, the local shadow service is enabled by default, and supports only unnamed classic shadows. You use the AWS IoT Greengrass Core SDK in your Lambda functions to interact with shadows on your devices.	In AWS IoT Greengrass V2, you enable the local shadow service by deploying the shadow manager component. • You can use the AWS IoT Device SDK V2 in Lambda functions and custom components to interact with shadows on your devices. • The local shadow service supports named shadows. • The local shadow service lets you delete shadows and synchronize deleted shadows with AWS IoT Core. For more information, see Interact with device shadows.

Concept	AWS IoT Greengrass V1	AWS IoT Greengrass V2

Subscriptions	In AWS IoT Greengrass V1, you define subscriptions for a Greengrass group to specify communication channels between Lambda functions, connectors, connected devices, the AWS IoT Core MQTT broker, and the local shadow service. Subscriptions specify where Lambda functions receive event messages to consume as function payloads.	In AWS IoT Greengrass V2, you specify communication channels without using subscriptions.
		• Components manage their own communication channels to interact with local publish/subscribe messages, AWS IoT Core MQTT messages, and the local shadow service.
		  • To develop a component that reacts to messages from another component or the AWS IoT Core MQTT broker, you can use interprocess communication (IPC) interfaces for local publish/subscribe messaging and AWS IoT Core MQTT messaging.
		  • To develop a component that interacts with the local shadow service, you can use the IPC interface for the local shadow service.
		  • In the component configuration, you define *authorization policies* to specify the topics and local shadows that the

Concept	AWS IoT Greengrass V1	AWS IoT Greengrass V2
		component has permissio n to use. • To configure communica tion channels between client devices, the local publish/subscribe broker, and the AWS IoT Core MQTT broker, you configure and deploy the MQTT bridge component. The MQTT bridge component enables you to interact with client devices in component s and relay messages between client devices and AWS IoT Core.
Accessing other AWS services	In AWS IoT Greengrass V1, you attach an AWS Identity and Access Management (IAM) role, called the group role, to a Greengrass group. The group role defines the permissions that Lambda functions and AWS IoT Greengrass features on that group's core device use to access AWS services.	In AWS IoT Greengrass V2, you attach an AWS IoT role alias to a Greengrass core device. The role alias points to an IAM role called the *token exchange role*. The token exchange role defines the permissions that Greengras s components on the core device use to access AWS services. For more informati on, see Authorize core devices to interact with AWS services.

Choose your runtime (Greengrass nucleus or Greengrass nucleus lite)

Choosing between Greengrass nucleus and Greengrass nucleus lite depends on your device resources and the features your Lambda functions use. Review the event source compatibility matrix in the following table, then use the decision flow diagram to determine which runtime is appropriate for your migration. For a detailed comparison of Greengrass nucleus and Greengrass nucleus lite features, see Choosing your runtime.

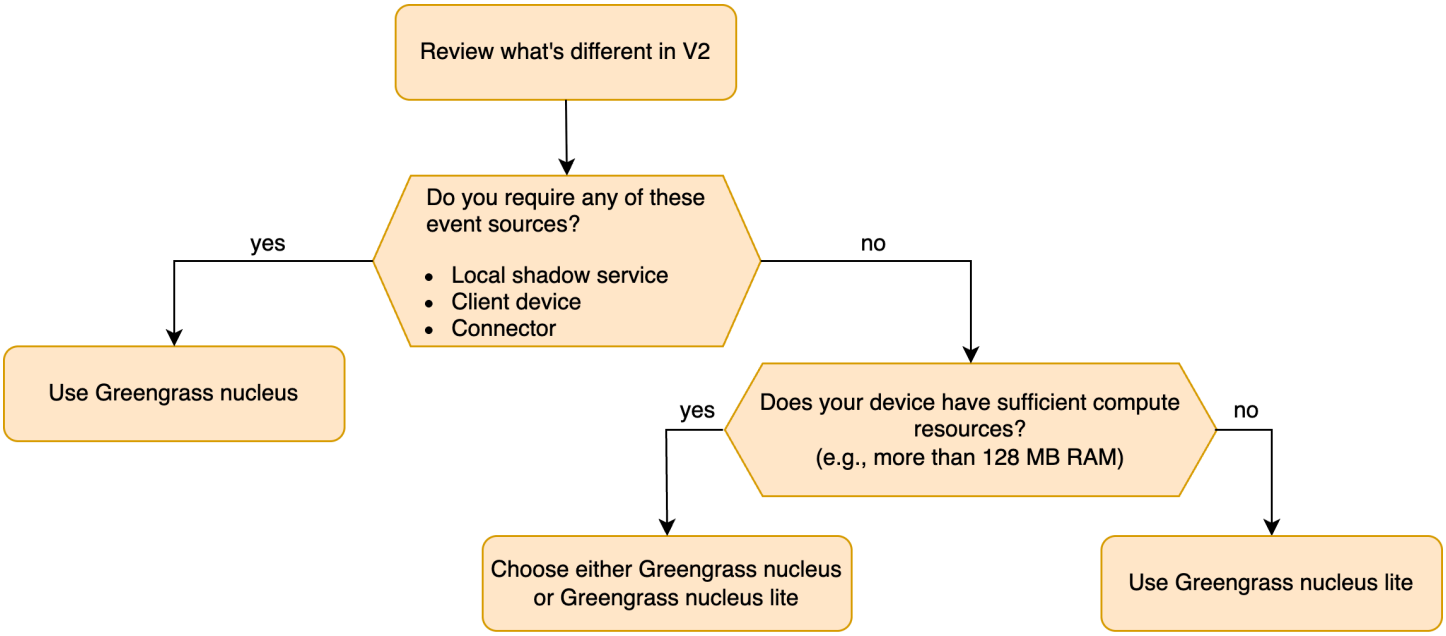
Event source compatibility matrix

In AWS IoT Greengrass V1, Lambda functions can communicate with five types of event sources: other Lambda functions, AWS IoT Core, local shadow service, client devices, and connectors. The following table shows which of these event sources are supported in each V2 runtime.

Note: Event source names use AWS IoT Greengrass V1 terminology. When migrating to V2, Lambda functions are converted to either Lambda components (supported only in Greengrass nucleus) or generic components (supported in both Greengrass nucleus and Greengrass nucleus lite).

Event Source	Greengrass nucleus	Greengrass nucleus lite
Other Lambda functions in the group	✓ (Lambda components and generic components)	✓ (Generic components only)
AWS IoT Core service	✓	✓
Local shadow service	✓	✗
Client device	✓	✗
Connector	✓	✗

Runtime selection decision flow



Notes

1. For Greengrass nucleus lite requirements and compatibility details, see Greengrass nucleus lite. Greengrass nucleus lite requires a minimum of 5 MB RAM and is designed for resource-constrained devices.
2. The decision flow provides guidance based on typical use cases, but is not a strict requirement. Customers with both resource-constrained and resource-sufficient devices may choose to use a single runtime across all devices for operational simplicity, even if some devices could support either runtime.

Next steps

After choosing your runtime, proceed to set up your test device:

- For Greengrass nucleus runtime: Set up a new device to test V1 applications on V2
- For Greengrass nucleus lite runtime: Set up a new device with Greengrass nucleus lite

Set up a new device to test V1 applications on V2

To minimize risk to your devices in production, create a new device to test your V1 applications on V2 before upgrading your production devices.

Choose one of the following setup guides based on your runtime selection from the previous step:

- **Option A - Greengrass nucleus runtime**: Follow Set up a new device to test V1 applications on V2 if you chose Greengrass nucleus. With this option, you can import Lambda functions as Lambda components with minimal code changes and supports V1 features like local shadow service, client devices, and connectors.
- **Option B - Greengrass nucleus lite runtime**: Follow Set up a new device to test V1 applications on V2 (Greengrass nucleus lite) if you chose Greengrass nucleus lite. This option requires converting Lambda functions to generic components using the AWS IoT Device SDK V2 or AWS IoT Greengrass Component SDK, but is optimized for resource-constrained devices.

Set up a new device to test V1 applications on V2

Set up a new AWS IoT Greengrass V2 core device to deploy and test AWS-provided components and AWS Lambda functions for your AWS IoT Greengrass V1 applications. You can also use this V2 core device to develop and test additional custom Greengrass components that run native processes on core devices. After you test your applications on a V2 core device, you can upgrade your existing V1 core devices to V2 and deploy the V2 components that provide your V1 functionality.

Step 1: Install AWS IoT Greengrass V2 on a new device

Install the AWS IoT Greengrass Core software v2.x on a new device. You can follow the getting started tutorial to set up a device and learn how to develop and deploy components. This tutorial uses automatic provisioning to quickly set up a device. When you install the AWS IoT Greengrass Core software v2.x, specify the `--deploy-dev-tools` argument to deploy the Greengrass CLI, so you can develop, test, and debug components directly on the device. For more information about other installation options, including how to install the AWS IoT Greengrass Core software behind a proxy or using a hardware security module (HSM), see Install the AWS IoT Greengrass Core software.

(Optional) Enable logging to Amazon CloudWatch Logs

To enable a V2 core device to upload logs to Amazon CloudWatch Logs, you can deploy the AWS-provided log manager component. You can use CloudWatch Logs to view component logs, so you can debug and troubleshoot without access to the core device's file system. For more information, see Monitor AWS IoT Greengrass logs.

Step 2: Create and deploy AWS IoT Greengrass V2 components to migrate AWS IoT Greengrass V1 applications

You can run most AWS IoT Greengrass V1 applications on AWS IoT Greengrass V2. You can import Lambda functions as components that run on AWS IoT Greengrass V2, and you can use AWS-provided components that offer the same functionality as AWS IoT Greengrass connectors.

You can also develop custom components to build any feature or runtime to run on Greengrass core devices. For information about how to develop and test components locally, see Create AWS IoT Greengrass components.

Topics

- Import V1 Lambda functions
- Use V1 connectors
- Run Docker containers
- Connect V1 Greengrass devices
- Enable the local shadow service
- Integrate with AWS IoT SiteWise

Import V1 Lambda functions

You can import Lambda functions as AWS IoT Greengrass V2 components. Choose from the following approaches:

- Import V1 Lambda functions directly as Greengrass components.
- Update your Lambda functions to use the Greengrass libraries in the AWS IoT Device SDK v2, and then import the Lambda functions as Greengrass components.
- Create custom components that use non-Lambda code and the AWS IoT Device SDK v2 to implement the same functionality as your Lambda functions.

If your Lambda function uses features, such as stream manager or local secrets, you must define dependencies on the AWS-provided components that package these features. When you deploy the Lambda function component, the deployment also includes the component for each feature that you define as a dependency. In the deployment, you can configure parameters, such as which secrets to deploy to the core device. Not all V1 features require a component dependency for your Lambda function on V2. The following list describes how to use V1 features in your V2 Lambda function component:

- **Access other AWS services**

  If your Lambda function uses AWS credentials to make requests to other AWS services, the core device's token exchange role must allow the core device to perform the AWS operations that the Lambda function uses. For more information, see Authorize core devices to interact with AWS services.

- **Stream manager**

  If your Lambda function uses stream manager, specify `aws.greengrass.StreamManager` as a component dependency when you import the function. When you deploy the stream manager component, specify the stream manager parameters to set for the target core devices. The core device's token exchange role must allow the core device to access the AWS Cloud destinations that you use with stream manager. For more information, see Stream manager.

- **Local secrets**

  If your Lambda function uses local secrets, specify `aws.greengrass.SecretManager` as a component dependency when you import the function. When you deploy the secret manager component, specify the secret resources to deploy to the target core devices. The core device's token exchange role must allow the core device to retrieve the secret resources to deploy. For more information, see Secret manager.

  When you deploy your Lambda function component, configure it to have an IPC authorization policy that grants permission to use the GetSecretValue IPC operation in the AWS IoT Device SDK V2.

- **Local shadows**

  If your Lambda function interacts with local shadows, you must update the Lambda function code to use the AWS IoT Device SDK V2. You must also specify `aws.greengrass.ShadowManager` as a component dependency when you import the function. For more information, see Interact with device shadows.

When you deploy your Lambda function component, configure it to have an IPC authorization policy that grants permission to use the shadow IPC operations in the AWS IoT Device SDK V2.

- **Subscriptions**
  - If your Lambda function subscribes to messages from a cloud source, specify those subscriptions as event sources when you import the function.
  - If your Lambda function subscribes to messages from another Lambda function, or if your Lambda function publishes messages to AWS IoT Core or other Lambda functions, configure and deploy the legacy subscription router component when you deploy your Lambda function. When you deploy the legacy subscription router component, specify the subscriptions that the Lambda function uses.

    > **Note**
    >
    > The legacy subscription router component is required only if your Lambda function uses the `publish()` function in the AWS IoT Greengrass Core SDK. If you update your Lambda function code to use the interprocess communication (IPC) interface in the AWS IoT Device SDK V2, you don't need to deploy the legacy subscription router component. For more information, see the following interprocess communication services:
    >
    > - Publish/subscribe local messages
    > - Publish/subscribe AWS IoT Core MQTT messages

  - If your Lambda function subscribes to messages from local connected devices, specify those subscriptions as event sources when you import the function. You must also configure and deploy the MQTT bridge component to relay messages from the connected devices to the local publish/subscribe topics that you specify as event sources.
  - If your Lambda function publishes messages to local connected devices, you must update the Lambda function code to use the AWS IoT Device SDK V2 to publish local publish/subscribe messages. You must also configure and deploy the MQTT bridge component to relay messages from the local publish/subscribe message broker to the connected devices.
- **Local volumes and devices**

  If your containerized Lambda function accesses local volumes or devices, specify those volumes and devices when you import the Lambda function. This feature doesn't require a component dependency.

For more information, see Run AWS Lambda functions.

Use V1 connectors

You can deploy AWS-provided components that offer the same functionality of some AWS IoT Greengrass connectors. When you create the deployment, you can configure the connectors' parameters.

The following AWS IoT Greengrass V2 components provide Greengrass V1 connector functionality:

- CloudWatch metrics component
- AWS IoT Device Defender component
- Firehose component
- Modbus-RTU protocol adapter component
- Amazon SNS component

Run Docker containers

AWS IoT Greengrass V2 doesn't provide a component to directly replace the V1 Docker application deployment connector. However, you can use the Docker application manager component to download Docker images, and then create custom components that run Docker containers from the downloaded images. For more information, see Run a Docker container and Docker application manager.

Connect V1 Greengrass devices

Connected devices in AWS IoT Greengrass V1 are called client devices in AWS IoT Greengrass V2. AWS IoT Greengrass V2 support for client devices is backward-compatible with AWS IoT Greengrass V1, so you can connect V1 client devices to V2 core devices without changing their application code. To enable client devices to connect to a V2 core device, deploy Greengrass components that enable client device support, and associate the client devices to the core device. To relay messages between client devices, the AWS IoT Core cloud service, and Greengrass components (including Lambda functions), deploy and configure the MQTT bridge component. You can deploy the IP detector component to automatically detect connectivity information, or you can manually manage endpoints. For more information, see Interact with local IoT devices.

Enable the local shadow service

In AWS IoT Greengrass V2, the local shadow service is implemented by the AWS-provided shadow manager component. AWS IoT Greengrass V2 also includes support for named shadows. To enable your components to interact with local shadows and to sync shadow states to AWS IoT Core, configure and deploy the shadow manager component, and use the shadow IPC operations in your component code. For more information, see Interact with device shadows.

Integrate with AWS IoT SiteWise

If you use your V1 core device as an AWS IoT SiteWise gateway, follow instructions to set up your new V2 core device as an AWS IoT SiteWise gateway. AWS IoT SiteWise provides an installation script that deploys the AWS IoT SiteWise components for you.

Step 3: Test your AWS IoT Greengrass V2 applications

After you create and deploy V2 components to your new V2 core device, verify that your applications meet your expectations. You can check the device's logs to view your components' standard output (stdout) and standard error (stderr) messages. For more information, see Monitor AWS IoT Greengrass logs.

If you deployed the Greengrass CLI to the core device, you can use it to debug components and their configurations. For more information, see Greengrass CLI commands.

After you verify that your applications work on a V2 core device, you can deploy your application's Greengrass components to other core devices. If you developed custom components that run native processes or Docker containers, you must first publish those components to the AWS IoT Greengrass service to deploy them to other core devices.

Set up a new device to test V1 applications on V2 (Greengrass nucleus lite)

Set up a new device with Greengrass nucleus lite to test generic components that you create to migrate your AWS IoT Greengrass V1 applications to V2. Greengrass nucleus lite is a lightweight runtime optimized for resource-constrained devices. You can use this device to develop and test custom Greengrass components that run native processes. After you test your applications on a Greengrass nucleus lite device, you can deploy the V2 components to other devices running Greengrass nucleus lite or upgrade your existing V1 core devices to V2.

Step 1: Install Greengrass nucleus lite on a new device

Install the Greengrass nucleus lite on a new device. Follow the Greengrass nucleus lite installation guide to set up a device.

> **Note**
>
> Greengrass nucleus lite does not support local shadow service, client devices, or connectors now. Ensure your V1 applications don't rely on these features before proceeding with this guide.

Step 2: Create and deploy generic components to migrate AWS IoT Greengrass V1 Lambda functions

To replicate the functionality of your AWS IoT Greengrass V1 Lambda functions on Greengrass nucleus lite, you must convert them to generic components. This involves rewriting your Lambda function code to use the AWS IoT Device SDK V2 or AWS IoT Greengrass Component SDK instead of the AWS IoT Greengrass Core SDK.

The following table lists the SDKs used in the V2 component examples in this guide:

SDK	Minimum version
AWS IoT Device SDK for Python v2	v1.11.3
AWS IoT Device SDK for Java v2	v1.9.3
AWS IoT Device SDK for JavaScript v2	v1.12.0
AWS IoT Greengrass Component SDK (C/C++) (currently in preview)	v0.4.0

Below examples show Lambda functions that use the AWS IoT Greengrass V1 Core SDK and their equivalent generic components, with component code, recipes, and build instructions in multiple programming languages for two main scenarios:

- **Local communication** - Components that communicate with other components on the same device using local pub/sub

- **Cloud communication** - Components that communicate with AWS IoT Core or other AWS services

Scenario 1: Local Communication (publisher → processor → subscriber)

This scenario demonstrates converting a V1 Lambda function that uses local pub/sub communication to a V2 generic component.

Application architecture

This example involves three components:

- Publisher Lambda publishes temperature data
- Processor Lambda receives data and processes it
- Processor Lambda publishes processed result to subscriber Lambda

The code example below focuses on the processor Lambda, which demonstrates both subscribing to and publishing messages for local communication.

V1 group subscriptions

In AWS IoT Greengrass V1, the following group subscriptions enable communication between Lambda functions:

Subscription 1: publisher → processor

- Source: Lambda (publisher)
- Target: Lambda (processor)
- Topic: sensors/temperature

Subscription 2: processor → subscriber

- Source: Lambda (processor)
- Target: Lambda (subscriber)
- Topic: lambda/alerts

Processor Lambda function (V1)

Python

```
import greengrasssdk
import json

iot_client = greengrasssdk.client('iot-data')

def lambda_handler(event, context):
    """
    Receives temperature from publisher Lambda,
    processes it, and forwards to subscriber Lambda
    """
    # Receive from publisher Lambda.
    sensor_id = event['sensor_id']
    temperature = event['temperature']

    print(f"Received from sensor {sensor_id}: {temperature}°F")

    # Process: Check if temperature is high.
    if temperature > 80:
        alert_data = {
            'sensor_id': sensor_id,
            'temperature': temperature,
            'alert': 'HIGH_TEMPERATURE'
        }

        # Publish to another Lambda using greengrasssdk.
        iot_client.publish(
            topic='lambda/alerts',
            payload=json.dumps(alert_data)
        )

        print(f"Alert sent to subscriber Lambda")

    return {'statusCode': 200}
```

Java

```
import com.amazonaws.greengrass.javasdk.IotDataClient;
import com.amazonaws.greengrass.javasdk.model.PublishRequest;
import com.amazonaws.services.lambda.runtime.Context;
```

```
import com.google.gson.Gson;
import java.nio.ByteBuffer;
import java.nio.charset.StandardCharsets;
import java.util.HashMap;
import java.util.Map;

public class TemperatureProcessorLambda {
    private static final Gson gson = new Gson();
    private final IotDataClient iotDataClient;

    public TemperatureProcessorLambda() {
        this.iotDataClient = new IotDataClient();
    }

    public String handleRequest(Map<String, Object> event, Context context) {
        /*
         * Receives temperature from publisher Lambda,
         * processes it, and forwards to subscriber Lambda
         */

        // Receive from publisher Lambda.
        String sensorId = (String) event.get("sensor_id");
        Number temp = (Number) event.get("temperature");
        int temperature = temp.intValue();

        System.out.println("Received from sensor " + sensorId + ": " + temperature +
 "°F");

        // Process: Check if temperature is high.
        if (temperature > 80) {
            Map<String, Object> alertData = new HashMap<>();
            alertData.put("sensor_id", sensorId);
            alertData.put("temperature", temperature);
            alertData.put("alert", "HIGH_TEMPERATURE");

            // Publish to another Lambda using greengrasssdk.
            String payload = gson.toJson(alertData);
            PublishRequest publishRequest = new PublishRequest()
                .withTopic("lambda/alerts")

 .withPayload(ByteBuffer.wrap(payload.getBytes(StandardCharsets.UTF_8)));

            iotDataClient.publish(publishRequest);
```

```
            System.out.println("Alert sent to subscriber Lambda");
        }

        return "Success";
    }
}
```

JavaScript

```
const greengrasssdk = require('aws-greengrass-core-sdk');

const iotClient = new greengrasssdk.IotData();

/**
 * Greengrass v1 Lambda function
 * Receives temperature from publisher Lambda,
 * processes it, and forwards to subscriber Lambda
 */
exports.handler = function(event, context) {
    // Receive from publisher Lambda.
    const sensorId = event.sensor_id;
    const temperature = event.temperature;

    console.log(`Received from sensor ${sensorId}: ${temperature}°F`);

    // Process: Check if temperature is high.
    if (temperature > 80) {
        const alertData = {
            sensor_id: sensorId,
            temperature: temperature,
            alert: 'HIGH_TEMPERATURE'
        };

        // Publish to another Lambda using greengrasssdk.
        const params = {
            topic: 'lambda/alerts',
            payload: JSON.stringify(alertData)
        };

        iotClient.publish(params, (err) => {
            if (err) {
                console.error('Error publishing alert:', err);
                context.fail(err);
```

```
            } else {
                console.log('Alert sent to subscriber Lambda');
                context.succeed('Success');
            }
        });
    } else {
        context.succeed('Success');
    }
};
```

C

```
#include <aws/greengrass/greengrasssdk.h>
#include <stdio.h>
#include <string.h>
#include <jansson.h>  // For JSON parsing.

static aws_greengrass_iot_data_client *iot_client = NULL;

void on_message_received(const char *topic, const uint8_t *payload, size_t
 payload_len, void *user_data) {
    // Parse the incoming message.
    char *payload_str = strndup((char *)payload, payload_len);
    json_error_t error;
    json_t *event = json_loads(payload_str, 0, &error);
    free(payload_str);

    if (!event) {
        fprintf(stderr, "Error parsing JSON: %s\n", error.text);
        return;
    }

    // Receive from publisher Lambda.
    json_t *sensor_id_obj = json_object_get(event, "sensor_id");
    json_t *temperature_obj = json_object_get(event, "temperature");

    const char *sensor_id = json_string_value(sensor_id_obj);
    int temperature = json_integer_value(temperature_obj);

    printf("Received from sensor %s: %d°F\n", sensor_id, temperature);

    // Process: Check if temperature is high.
    if (temperature > 80) {
```

```
        // Create alert data.
        json_t *alert_data = json_object();
        json_object_set_new(alert_data, "sensor_id", json_string(sensor_id));
        json_object_set_new(alert_data, "temperature", json_integer(temperature));
        json_object_set_new(alert_data, "alert", json_string("HIGH_TEMPERATURE"));

        // Convert to JSON string.
        char *alert_payload = json_dumps(alert_data, JSON_COMPACT);

        // Publish to another Lambda using greengrasssdk.
        aws_greengrass_publish_params params = {
            .topic = "lambda/alerts",
            .payload = (uint8_t *)alert_payload,
            .payload_len = strlen(alert_payload)
        };

        aws_greengrass_iot_data_publish(iot_client, &params);

        printf("Alert sent to subscriber Lambda\n");

        free(alert_payload);
        json_decref(alert_data);
    }

    json_decref(event);
}

int main(int argc, char *argv[]) {
    // Initialize Greengrass SDK.
    iot_client = aws_greengrass_iot_data_client_new();

    // Subscribe to temperature sensor topic.
    aws_greengrass_subscribe_params subscribe_params = {
        .topic = "sensors/temperature",
        .callback = on_message_received,
        .user_data = NULL
    };

    aws_greengrass_iot_data_subscribe(iot_client, &subscribe_params);

    printf("Temperature Processor Lambda started\n");
    printf("Subscribed to sensors/temperature\n");
    printf("Waiting for sensor data...\n");
```

```
    // Keep the Lambda running.
    while (1) {
        sleep(1);
    }

    return 0;
}
```

C++

```
#include <aws/greengrass/greengrasssdk.h>
#include <iostream>
#include <string>
#include <memory>
#include <jansson.h> // For JSON parsing.
#include <unistd.h>

class TemperatureProcessor {
private:
    std::unique_ptr<aws_greengrass_iot_data_client,
                    decltype(&aws_greengrass_iot_data_client_destroy)> iot_client;

    static void message_callback_wrapper(const char *topic,
                                         const uint8_t *payload,
                                         size_t payload_len,
                                         void *user_data) {
        auto* processor = static_cast<TemperatureProcessor*>(user_data);
        processor->on_message_received(topic, payload, payload_len);
    }

public:
    TemperatureProcessor()
        : iot_client(aws_greengrass_iot_data_client_new(),
                     aws_greengrass_iot_data_client_destroy) {
        if (!iot_client) {
            throw std::runtime_error("Failed to create Greengrass IoT client");
        }
    }

    void on_message_received(const char *topic,
                             const uint8_t *payload,
                             size_t payload_len) {
        // Parse the incoming message.
```

```
        std::string payload_str(reinterpret_cast<const char*>(payload),
payload_len);

        json_error_t error;
        json_t *event = json_loads(payload_str.c_str(), 0, &error);

        if (!event) {
            std::cerr << "Error parsing JSON: " << error.text << std::endl;
            return;
        }

        json_t *sensor_id_obj = json_object_get(event, "sensor_id");
        json_t *temperature_obj = json_object_get(event, "temperature");

        const char *sensor_id = json_string_value(sensor_id_obj);
        int temperature = json_integer_value(temperature_obj);

        std::cout << "Received from sensor " << sensor_id
                  << ": " << temperature << "°F" << std::endl;

        if (temperature > 80) {
            send_alert(sensor_id, temperature);
        }

        json_decref(event);
    }

    void send_alert(const char *sensor_id, int temperature) {
        // Create alert data.
        json_t *alert_data = json_object();
        json_object_set_new(alert_data, "sensor_id", json_string(sensor_id));
        json_object_set_new(alert_data, "temperature", json_integer(temperature));
        json_object_set_new(alert_data, "alert", json_string("HIGH_TEMPERATURE"));

        // Convert to JSON string.
        char *alert_payload = json_dumps(alert_data, JSON_COMPACT);

        // Publish to another Lambda using greengrasssdk.
        aws_greengrass_publish_params params = {
            .topic = "lambda/alerts",
            .payload = reinterpret_cast<uint8_t*>(alert_payload),
            .payload_len = strlen(alert_payload)
        };
```

```
        aws_greengrass_iot_data_publish(iot_client.get(), &params);

        std::cout << "Alert sent to subscriber Lambda" << std::endl;

        free(alert_payload);
        json_decref(alert_data);
    }

    void subscribe_to_topic(const std::string& topic) {
        aws_greengrass_subscribe_params subscribe_params = {
            .topic = topic.c_str(),
            .callback = message_callback_wrapper,
            .user_data = this
        };

        aws_greengrass_iot_data_subscribe(iot_client.get(), &subscribe_params);

        std::cout << "Temperature Processor Lambda started" << std::endl;
        std::cout << "Subscribed to " << topic << std::endl;
        std::cout << "Waiting for sensor data..." << std::endl;
    }

    void run() {
        // Keep the Lambda running.
        while (true) {
            sleep(1);
        }
    }
};

int main(int argc, char *argv[]) {
    try {
        TemperatureProcessor processor;
        processor.subscribe_to_topic("sensors/temperature");
        processor.run();
    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << std::endl;
        return 1;
    }

    return 0;
}
```

Generic component (V2)

To achieve the same functionality in AWS IoT Greengrass V2, create a generic component with the following:

1. Component code

Python

Prerequisites: Before using this component code, install and verify the AWS IoT Device SDK for Python on your Greengrass device:

```
# Install the SDK
pip3 install awsiotsdk

# Verify installation
python3 -c "import awsiot.greengrasscoreipc.clientv2; print('SDK installed
 successfully')"
```

If you encounter dependency conflicts during installation, try installing a specific version of the AWS IoT Device SDK.

If the verification command prints "SDK installed successfully", you're ready to use the component code:

```
from awsiot.greengrasscoreipc.clientv2 import GreengrassCoreIPCClientV2
from awsiot.greengrasscoreipc.model import (
    PublishMessage,
    JsonMessage
)
import time

ipc_client = GreengrassCoreIPCClientV2()

def on_sensor_data(event):
    """
    Receives temperature from sensor publisher component,
    processes it, and forwards to alert component
    """
    try:
        # Receive from publisher component.
        data = event.json_message.message
        sensor_id = data['sensor_id']
```

```
        temperature = data['temperature']

        print(f"Received from sensor {sensor_id}: {temperature}°F")

        # Process: Check if temperature is high.
        if temperature > 80:
            alert_data = {
                'sensor_id': sensor_id,
                'temperature': temperature,
                'alert': 'HIGH_TEMPERATURE'
            }

            # Publish to another component (AlertHandler).
            ipc_client.publish_to_topic(
                topic='component/alerts',
                publish_message=PublishMessage(
                    json_message=JsonMessage(message=alert_data)
                )
            )

            print(f"Alert sent to AlertHandler component")

    except Exception as e:
        print(f"Error processing sensor data: {e}")

def main():
    print("Temperature Processor component starting...")

    # Subscribe to sensor data from publisher component.
    ipc_client.subscribe_to_topic(
        topic='sensors/temperature',
        on_stream_event=on_sensor_data
    )

    print("Subscribed to sensors/temperature")
    print("Waiting for sensor data...")

    # Keep running.
    while True:
        time.sleep(1)

if __name__ == '__main__':
    main()
```

Java

```
import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClientV2;
import software.amazon.awssdk.aws.greengrass.model.PublishMessage;
import software.amazon.awssdk.aws.greengrass.model.PublishToTopicRequest;
import software.amazon.awssdk.aws.greengrass.model.JsonMessage;
import software.amazon.awssdk.aws.greengrass.model.SubscribeToTopicRequest;
import software.amazon.awssdk.aws.greengrass.model.SubscriptionResponseMessage;
import java.util.HashMap;
import java.util.Map;
import java.util.Optional;

public class TemperatureProcessor {
    private static GreengrassCoreIPCClientV2 ipcClient;

    public static void main(String[] args) {
        System.out.println("Temperature Processor component starting...");

        try (GreengrassCoreIPCClientV2 client =
 GreengrassCoreIPCClientV2.builder().build()) {
            ipcClient = client;

            SubscribeToTopicRequest subscribeRequest = new SubscribeToTopicRequest()
                .withTopic("sensors/temperature");

            ipcClient.subscribeToTopic(
                subscribeRequest,
                TemperatureProcessor::onSensorData,
                Optional.empty(),
                Optional.empty()
            );

            System.out.println("Subscribed to sensors/temperature");
            System.out.println("Waiting for sensor data...");

            while (true) {
                Thread.sleep(1000);
            }
        } catch (Exception e) {
            System.err.println("Error: " + e.getMessage());
            e.printStackTrace();
        }
    }
```

```
    public static void onSensorData(SubscriptionResponseMessage message) {
        try {
            Map<String, Object> data = message.getJsonMessage().getMessage();
            String sensorId = (String) data.get("sensor_id");
            Number temp = (Number) data.get("temperature");
            int temperature = temp.intValue();

            System.out.println("Received from sensor " + sensorId + ": " +
 temperature + "F");

            if (temperature > 80) {
                Map<String, Object> alertData = new HashMap<>();
                alertData.put("sensor_id", sensorId);
                alertData.put("temperature", temperature);
                alertData.put("alert", "HIGH_TEMPERATURE");

                JsonMessage jsonMessage = new JsonMessage().withMessage(alertData);
                PublishMessage publishMessage = new
 PublishMessage().withJsonMessage(jsonMessage);
                PublishToTopicRequest publishRequest = new PublishToTopicRequest()
                    .withTopic("component/alerts")
                    .withPublishMessage(publishMessage);

                ipcClient.publishToTopic(publishRequest);
                System.out.println("Alert sent to AlertHandler component");
            }
        } catch (Exception e) {
            System.err.println("Error processing sensor data: " + e.getMessage());
        }
    }
}
```

JavaScript

```
const greengrasscoreipc = require('aws-iot-device-sdk-v2').greengrasscoreipc;

class TemperatureProcessor {
    constructor() {
        this.ipcClient = null;
    }

    async start() {
        console.log('Temperature Processor component starting...');
```

```
        try {
            this.ipcClient = greengrasscoreipc.createClient();
            await this.ipcClient.connect();

            const subscribeRequest = {
                topic: 'sensors/temperature'
            };

            const streamingOperation =
this.ipcClient.subscribeToTopic(subscribeRequest);

            streamingOperation.on('message', (message) => {
                this.onSensorData(message);
            });

            streamingOperation.on('streamError', (error) => {
                console.error('Stream error:', error);
            });

            streamingOperation.on('ended', () => {
                console.log('Subscription stream ended');
            });

            await streamingOperation.activate();

            console.log('Subscribed to sensors/temperature');
            console.log('Waiting for sensor data...');

        } catch (error) {
            console.error('Error starting component:', error);
            process.exit(1);
        }
    }

    async onSensorData(message) {
        try {
            const data = message.jsonMessage.message;

            const sensorId = data.sensor_id;
            const temperature = data.temperature;

            console.log(`Received from sensor ${sensorId}: ${temperature}°F`);
```

```
            if (temperature > 80) {
                const alertData = {
                    sensor_id: sensorId,
                    temperature: temperature,
                    alert: 'HIGH_TEMPERATURE'
                };

                const publishRequest = {
                    topic: 'component/alerts',
                    publishMessage: {
                        jsonMessage: {
                            message: alertData
                        }
                    }
                };

                await this.ipcClient.publishToTopic(publishRequest);
                console.log('Alert sent to AlertHandler component');
            }

        } catch (error) {
            console.error('Error processing sensor data:', error);
        }
    }
}

// Start the component.
const processor = new TemperatureProcessor();
processor.start();
```

C

```
#include <gg/buffer.h>
#include <gg/error.h>
#include <gg/ipc/client.h>
#include <gg/map.h>
#include <gg/object.h>
#include <gg/sdk.h>
#include <unistd.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <pthread.h>

#define SUBSCRIBE_TOPIC "sensors/temperature"
#define PUBLISH_TOPIC "component/alerts"

typedef struct {
    char sensor_id[64];
    int64_t temperature;
} AlertData;

static pthread_mutex_t alert_mutex = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t alert_cond = PTHREAD_COND_INITIALIZER;
static AlertData pending_alert;
static bool has_pending_alert = false;

static void *alert_publisher_thread(void *arg) {
    (void) arg;

    while (true) {
        pthread_mutex_lock(&alert_mutex);
        while (!has_pending_alert) {
            pthread_cond_wait(&alert_cond, &alert_mutex);
        }

        AlertData alert = pending_alert;
        has_pending_alert = false;
        pthread_mutex_unlock(&alert_mutex);

        GgBuffer sensor_id_buf = { .data = (uint8_t *)alert.sensor_id, .len =
 strlen(alert.sensor_id) };
        GgMap payload = GG_MAP(
            gg_kv(GG_STR("sensor_id"), gg_obj_buf(sensor_id_buf)),
            gg_kv(GG_STR("temperature"), gg_obj_i64(alert.temperature)),
            gg_kv(GG_STR("alert"), gg_obj_buf(GG_STR("HIGH_TEMPERATURE")))
        );

        GgError ret = ggipc_publish_to_topic_json(GG_STR(PUBLISH_TOPIC), payload);

        if (ret != GG_ERR_OK) {
            fprintf(stderr, "Failed to publish alert\n");
        } else {
            printf("Alert sent to AlertHandler component\n");
        }
    }
```

```
    return NULL;
}

static void on_sensor_data(
    void *ctx, GgBuffer topic, GgObject payload, GgIpcSubscriptionHandle handle
) {
    (void) ctx;
    (void) topic;
    (void) handle;

    if (gg_obj_type(payload) != GG_TYPE_MAP) {
        fprintf(stderr, "Expected JSON message\n");
        return;
    }

    GgMap map = gg_obj_into_map(payload);

    GgObject *sensor_id_obj;
    if (!gg_map_get(map, GG_STR("sensor_id"), &sensor_id_obj)) {
        fprintf(stderr, "Missing sensor_id field\n");
        return;
    }
    GgBuffer sensor_id = gg_obj_into_buf(*sensor_id_obj);

    GgObject *temperature_obj;
    if (!gg_map_get(map, GG_STR("temperature"), &temperature_obj)) {
        fprintf(stderr, "Missing temperature field\n");
        return;
    }
    int64_t temperature = gg_obj_into_i64(*temperature_obj);

    printf("Received from sensor %.*s: %lld°F\n",
           (int)sensor_id.len, sensor_id.data, (long long)temperature);

    if (temperature > 80) {
        pthread_mutex_lock(&alert_mutex);
        snprintf(pending_alert.sensor_id, sizeof(pending_alert.sensor_id),
                 "%.*s", (int)sensor_id.len, sensor_id.data);
        pending_alert.temperature = temperature;
        has_pending_alert = true;
        pthread_cond_signal(&alert_cond);
        pthread_mutex_unlock(&alert_mutex);
    }
```

```
}

int main(void) {
    setvbuf(stdout, NULL, _IONBF, 0);
    printf("Temperature Processor component starting...\n");

    gg_sdk_init();

    GgError ret = ggipc_connect();
    if (ret != GG_ERR_OK) {
        fprintf(stderr, "Failed to connect to Greengrass nucleus\n");
        exit(1);
    }
    printf("Connected to Greengrass IPC\n");

    // Start alert publisher thread.
    pthread_t alert_thread;
    if (pthread_create(&alert_thread, NULL, alert_publisher_thread, NULL) != 0) {
        fprintf(stderr, "Failed to create alert publisher thread\n");
        exit(1);
    }

    GgIpcSubscriptionHandle handle;
    ret = ggipc_subscribe_to_topic(
        GG_STR(SUBSCRIBE_TOPIC),
        &on_sensor_data,
        NULL,
        &handle
    );

    if (ret != GG_ERR_OK) {
        fprintf(stderr, "Failed to subscribe to topic\n");
        exit(1);
    }

    printf("Subscribed to %s\n", SUBSCRIBE_TOPIC);
    printf("Waiting for sensor data...\n");

    // Keep running.
    while (true) {
        sleep(1);
    }

    return 0;
```

```
}
```

C++

```
#include <gg/ipc/client.hpp>
#include <gg/buffer.hpp>
#include <gg/object.hpp>
#include <gg/types.hpp>

#include <chrono>
#include <condition_variable>
#include <iostream>
#include <mutex>
#include <string>
#include <string_view>
#include <thread>

struct AlertData {
    std::string sensor_id;
    int64_t temperature;
};

static std::mutex alert_mutex;
static std::condition_variable alert_cv;
static AlertData pending_alert;
static bool has_pending_alert = false;

void alert_publisher_thread() {
    auto& client = gg::ipc::Client::get();

    while (true) {
        std::unique_lock<std::mutex> lock(alert_mutex);
        alert_cv.wait(lock, [] { return has_pending_alert; });

        AlertData alert = pending_alert;
        has_pending_alert = false;
        lock.unlock();

        // Create alert payload as JSON string.
        std::string json_payload = "{\"sensor_id\":\"" + alert.sensor_id +
                                   "\",\"temperature\":" +
 std::to_string(alert.temperature) +
                                   ",\"alert\":\"HIGH_TEMPERATURE\"}";
```

```
        // Convert to Buffer and publish.
        gg::Buffer buffer(json_payload);
        auto error = client.publish_to_topic("component/alerts", buffer);

        if (error) {
            std::cerr << "Failed to publish alert\n";
        } else {
            std::cout << "Alert sent to AlertHandler component\n";
        }
    }
}

class SensorCallback : public gg::ipc::LocalTopicCallback {
    void operator()(
        std::string_view topic,
        gg::Object payload,
        gg::ipc::Subscription& handle
    ) override {
        (void) topic;
        (void) handle;

        // Payload is a Buffer containing JSON string.
        if (payload.index() != GG_TYPE_BUF) {
            std::cerr << "Expected Buffer message\n";
            return;
        }

        // Extract buffer using gg::get.
        auto buffer = gg::get<std::span<uint8_t>>(payload);
        std::string json_str(reinterpret_cast<const char*>(buffer.data()),
 buffer.size());

        // Simple JSON parsing for demo.
        std::string sensor_id = "sensor1";
        int64_t temperature = 0;

        // Extract temperature (simple string search).
        size_t temp_pos = json_str.find("\"temperature\":");
        if (temp_pos != std::string::npos) {
            temp_pos += 14; // Skip "temperature".
            size_t end_pos = json_str.find_first_of(",}", temp_pos);
            if (end_pos != std::string::npos) {
```

```
                temperature = std::stoll(json_str.substr(temp_pos, end_pos -
 temp_pos));
            }
        }

        std::cout << "Received from sensor " << sensor_id << ": "
                  << temperature << "°F\n";

        if (temperature > 80) {
            std::lock_guard<std::mutex> lock(alert_mutex);
            pending_alert = {sensor_id, temperature};
            has_pending_alert = true;
            alert_cv.notify_one();
        }
    }
};

int main() {
    // Disable stdout buffering for real-time logging.
    std::cout.setf(std::ios::unitbuf);

    std::cout << "Temperature Processor component starting..." << std::endl;

    auto& client = gg::ipc::Client::get();
    std::cout << "Got client instance" << std::endl;

    auto error = client.connect();
    std::cout << "Connect returned, error code: " << error.value() << std::endl;

    if (error) {
        std::cerr << "Failed to connect to Greengrass nucleus: " << error.message()
 << std::endl;
        return 1;
    }

    std::cout << "Connected to Greengrass IPC" << std::endl;

    // Start alert publisher thread.
    std::thread alert_thread(alert_publisher_thread);
    alert_thread.detach();

    // Handler must be static lifetime if subscription handle is not held.
    static SensorCallback handler;
    error = client.subscribe_to_topic("sensors/temperature", handler);
```

```
    if (error) {
        std::cerr << "Failed to subscribe to topic: " << error.message() <<
 std::endl;
        return 1;
    }

    std::cout << "Subscribed to sensors/temperature" << std::endl;
    std::cout << "Waiting for sensor data..." << std::endl;

    // Keep running.
    while (true) {
        using namespace std::chrono_literals;
        std::this_thread::sleep_for(1s);
    }

    return 0;
}
```

2. Build and package the component

Some languages require building or packaging before deployment.

Python

Python does not require compilation. The component can use the .py file directly.

Java

To build an executable JAR with all dependencies bundled:

1. Create a `pom.xml` file in your project directory:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <!-- Basic project information: organization, component name, and version --
>
    <groupId>com.example</groupId>
```

```
    <artifactId>temperature-processor</artifactId>
    <version>1.0.0</version>

    <properties>
        <!-- Java 11 LTS (Long Term Support) is recommended for Greengrass v2
components -->
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
    </properties>

    <dependencies>
        <!-- AWS IoT Device SDK for Java - provides IPC client for Greengrass v2
local communication -->
        <dependency>
            <groupId>software.amazon.awssdk.iotdevicesdk</groupId>
            <artifactId>aws-iot-device-sdk</artifactId>
            <version>1.25.1</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <!-- Maven Shade Plugin - creates a standalone JAR with all
dependencies included for Greengrass deployment -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>3.2.4</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                        <configuration>
                            <transformers>
                                <!-- Set the main class for the executable JAR
-->
                                <transformer
implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                                    <mainClass>TemperatureProcessor</mainClass>
                                </transformer>
                            </transformers>
                            <filters>
```

```
                                <!-- Exclude signature files to avoid security
 exceptions -->
                                <filter>
                                    <artifact>*:*</artifact>
                                    <excludes>
                                        <exclude>META-INF/*.SF</exclude>
                                        <exclude>META-INF/*.DSA</exclude>
                                        <exclude>META-INF/*.RSA</exclude>
                                    </excludes>
                                </filter>
                            </filters>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

2. Build the JAR:

```
mvn clean package
```

This creates `target/temperature-processor-1.0.0.jar` with all dependencies included.

3. Upload the JAR to your S3 bucket for deployment.

JavaScript

To package your Node.js component with all dependencies:

1. Create a `package.json` file:

```
{
  "name": "temperature-processor",
  "version": "1.0.0",
  "description": "Temperature processor component for Greengrass v2",
  "main": "TemperatureProcessor.js",
  "dependencies": {
    "aws-iot-device-sdk-v2": "^1.21.0"
  },
  "engines": {
```

```
    "node": ">=14.0.0"
  }
}
```

2. Install dependencies on your development machine:

```
npm install
```

   This creates a `node_modules` folder containing the AWS AWS IoT Device SDK v2.

3. Package for deployment:

```
zip -r TemperatureProcessor.zip TemperatureProcessor.js node_modules/
 package.json
```

4. Upload the zip file to your S3 bucket for deployment.

> **Note**
>
> The Greengrass device must have Node.js runtime installed (version 14 or later). You don't need to run `npm install` on the Greengrass core device since the component artifact includes all dependencies in the bundled `node_modules` folder.

C

Prerequisites:

To build the SDK and the component, you will need the following build dependencies:

- GCC or Clang
- CMake (at least version 3.22)
- Make or Ninja

Install build dependencies:

On Ubuntu/Debian:

```
sudo apt install build-essential cmake
```

On Amazon Linux:

```
sudo yum install gcc cmake make
```

Create a CMakeLists.txt file for your component:

```
cmake_minimum_required(VERSION 3.10)
project(TemperatureProcessor C)

set(CMAKE_C_STANDARD 11)

# Add AWS Greengrass Component SDK
add_subdirectory(aws-greengrass-component-sdk)

# Build your component executable
add_executable(temperature_processor temperature_processor.c)
target_link_libraries(temperature_processor gg-sdk)
```

Build steps:

```
# Clone the AWS Greengrass Component SDK into your project
git clone https://github.com/aws-greengrass/aws-greengrass-component-sdk.git

# Build your component
cmake -B build -D CMAKE_BUILD_TYPE=MinSizeRel
make -C build -j$(nproc)

# The binary 'temperature_processor' is in ./build/
# Upload this binary to S3 for deployment
```

C++

Prerequisites:

To build the SDK and the component, you will need the following build dependencies:

- GCC or Clang with C++20 support
- CMake (at least version 3.22)
- Make or Ninja

Install build dependencies:

On Ubuntu/Debian:

```
sudo apt install build-essential cmake
```

On Amazon Linux:

```
sudo yum install gcc-c++ cmake make
```

Create a CMakeLists.txt file for your component:

```
cmake_minimum_required(VERSION 3.10)
project(TemperatureProcessor CXX)

set(CMAKE_CXX_STANDARD 20)

# Add SDK as subdirectory
add_subdirectory(aws-greengrass-component-sdk)

# Add C++ SDK subdirectory
add_subdirectory(aws-greengrass-component-sdk/cpp)

add_executable(temperature_processor temperature_processor.cpp)
target_link_libraries(temperature_processor gg-sdk++)
```

Build steps:

```
# Clone the AWS Greengrass Component SDK into your project
git clone https://github.com/aws-greengrass/aws-greengrass-component-sdk.git

# Build your component
cmake -B build -D CMAKE_BUILD_TYPE=MinSizeRel
make -C build -j$(nproc)

# The binary 'temperature_processor' will be in ./build/
# Upload this binary to S3 for deployment
```

3. Component recipe

Update the "resources" array with the actual topics your component uses.

Python

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.TemperatureProcessor",
  "ComponentVersion": "1.0.0",
  "ComponentType": "aws.greengrass.generic",
  "ComponentDescription": "Receives sensor data and forwards alerts to
 AlertHandler",
  "ComponentPublisher": "[Your Company]",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
          "com.example.TemperatureProcessor:pubsub:1": {
            "policyDescription": "Allows access to subscribe to sensor topics",
            "operations": [
              "aws.greengrass#SubscribeToTopic"
            ],
            "resources": [
              "sensors/temperature"
            ]
          },
          "com.example.TemperatureProcessor:pubsub:2": {
            "policyDescription": "Allows access to publish to alert topics",
            "operations": [
              "aws.greengrass#PublishToTopic"
            ],
            "resources": [
              "component/alerts"
            ]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux",
        "runtime": "*"
      },
      "Lifecycle": {
        "run": "python3 -u {artifacts:path}/temperature_processor.py"
```

```
      },
      "Artifacts": [
        {
          "Uri": "s3://YOUR-BUCKET/artifacts/com.example.TemperatureProcessor/1.0.0/
temperature_processor.py"
        }
      ]
    }
  ]
}
```

Java

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.TemperatureProcessor",
  "ComponentVersion": "1.0.0",
  "ComponentType": "aws.greengrass.generic",
  "ComponentDescription": "Receives sensor data and forwards alerts to
 AlertHandler",
  "ComponentPublisher": "[Your Company]",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
          "com.example.TemperatureProcessor:pubsub:1": {
            "policyDescription": "Allows access to subscribe to sensor topics",
            "operations": [
              "aws.greengrass#SubscribeToTopic"
            ],
            "resources": [
              "sensors/temperature"
            ]
          },
          "com.example.TemperatureProcessor:pubsub:2": {
            "policyDescription": "Allows access to publish to alert topics",
            "operations": [
              "aws.greengrass#PublishToTopic"
            ],
            "resources": [
              "component/alerts"
            ]
          }
```

```
        }
      }
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux",
        "runtime": "*"
      },
      "Lifecycle": {
        "run": "java -jar {artifacts:path}/TemperatureProcessor.jar"
      },
      "Artifacts": [
        {
          "Uri": "s3://YOUR-BUCKET/artifacts/com.example.TemperatureProcessor/1.0.0/
TemperatureProcessor.jar"
        }
      ]
    }
  ]
}
```

JavaScript

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.TemperatureProcessor",
  "ComponentVersion": "1.0.0",
  "ComponentType": "aws.greengrass.generic",
  "ComponentDescription": "Receives sensor data and forwards alerts to
 AlertHandler",
  "ComponentPublisher": "[Your Company]",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
          "com.example.TemperatureProcessor:pubsub:1": {
            "policyDescription": "Allows access to subscribe to sensor topics",
            "operations": [
              "aws.greengrass#SubscribeToTopic"
            ],
            "resources": [
```

```
              "sensors/temperature"
            ]
          },
          "com.example.TemperatureProcessor:pubsub:2": {
            "policyDescription": "Allows access to publish to alert topics",
            "operations": [
              "aws.greengrass#PublishToTopic"
            ],
            "resources": [
              "component/alerts"
            ]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux",
        "runtime": "*"
      },
      "Lifecycle": {
        "run": "cd {artifacts:decompressedPath}/TemperatureProcessor && node
 TemperatureProcessor.js"
      },
      "Artifacts": [
        {
          "Uri": "s3://YOUR-BUCKET/artifacts/com.example.TemperatureProcessor/1.0.0/
TemperatureProcessor.zip",
          "Unarchive": "ZIP"
        }
      ]
    }
  ]
}
```

C/C++

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.TemperatureProcessor",
  "ComponentVersion": "1.0.0",
```

```
  "ComponentType": "aws.greengrass.generic",
  "ComponentDescription": "Receives sensor data and forwards alerts to
 AlertHandler",
  "ComponentPublisher": "[Your Company]",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
          "com.example.TemperatureProcessor:pubsub:1": {
            "policyDescription": "Allows access to subscribe to sensor topics",
            "operations": [
              "aws.greengrass#SubscribeToTopic"
            ],
            "resources": [
              "sensors/temperature"
            ]
          },
          "com.example.TemperatureProcessor:pubsub:2": {
            "policyDescription": "Allows access to publish to alert topics",
            "operations": [
              "aws.greengrass#PublishToTopic"
            ],
            "resources": [
              "component/alerts"
            ]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux",
        "runtime": "*"
      },
      "Lifecycle": {
        "run": "{artifacts:path}/temperature_processor"
      },
      "Artifacts": [
        {
          "Uri": "s3://YOUR-BUCKET/artifacts/com.example.TemperatureProcessor/1.0.0/
temperature_processor"
        }
```

```
      ]
    }
  ]
}
```

Scenario 2: Cloud Communication

This scenario demonstrates converting a V1 Lambda function that communicates with AWS IoT Core to a V2 generic component.

Application architecture

This example uses a cloud-connected architecture:

- AWS IoT Core sends commands to the device
- Controller Lambda receives commands and processes them
- Controller Lambda sends telemetry data back to AWS IoT Core

This example focuses on the controller Lambda, which demonstrates both receiving messages from and publishing messages to AWS IoT Core.

V1 group subscriptions

In AWS IoT Greengrass V1, the following group subscriptions enable communication between the Lambda function and AWS IoT Core:

Subscription 1: IoT Cloud → Lambda

- Source: IoT Cloud
- Target: Lambda (DeviceController)
- Topic: commands/device1

Subscription 2: Lambda → IoT Cloud

- Source: Lambda (DeviceController)
- Target: IoT Cloud
- Topic: telemetry/device1

Controller Lambda function (V1)

Python

```
import greengrasssdk
import json
import time

iot_client = greengrasssdk.client('iot-data')

def lambda_handler(event, context):
    """
    Receives commands from IoT Core,
    processes them, and sends telemetry back to cloud
    """
    # Receive command from IoT Core.
    command = event.get('command')
    device_id = event.get('device_id', 'device1')

    print(f"Received command from cloud: {command}")

    # Process command.
    if command == 'get_status':
        status = get_device_status()

        # Send telemetry back to IoT Core.
        telemetry_data = {
            'device_id': device_id,
            'status': status,
            'timestamp': time.time()
        }

        iot_client.publish(
            topic=f'telemetry/{device_id}',
            payload=json.dumps(telemetry_data)
        )

        print(f"Telemetry sent to cloud: {telemetry_data}")

    return {'statusCode': 200}

def get_device_status():
    """Get current device status"""
    # Simulate getting device status.
```

```
    return 'online'
```

Java

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.greengrass.javasdk.IotDataClient;
import com.amazonaws.greengrass.javasdk.model.PublishRequest;
import com.google.gson.Gson;
import java.nio.ByteBuffer;
import java.nio.charset.StandardCharsets;
import java.util.HashMap;
import java.util.Map;

public class DeviceControllerLambda {
    private static final Gson gson = new Gson();
    private final IotDataClient iotDataClient;

    public DeviceControllerLambda() {
        this.iotDataClient = new IotDataClient();
    }

    public String handleRequest(Map<String, Object> event, Context context) {
        /*
         * Receives commands from IoT Core,
         * processes them, and sends telemetry back to cloud
         */

        // Receive command from IoT Core.
        String command = (String) event.get("command");
        String deviceId = event.containsKey("device_id") ?
            (String) event.get("device_id") : "device1";

        System.out.println("Received command from cloud: " + command);

        // Process command.
        if ("get_status".equals(command)) {
            String status = getDeviceStatus();

            // Send telemetry back to IoT Core.
            Map<String, Object> telemetryData = new HashMap<>();
            telemetryData.put("device_id", deviceId);
            telemetryData.put("status", status);
            telemetryData.put("timestamp", System.currentTimeMillis() / 1000.0);
```

```
            String payload = gson.toJson(telemetryData);
            PublishRequest publishRequest = new PublishRequest()
                .withTopic("telemetry/" + deviceId)

 .withPayload(ByteBuffer.wrap(payload.getBytes(StandardCharsets.UTF_8)));

            iotDataClient.publish(publishRequest);

            System.out.println("Telemetry sent to cloud: " + telemetryData);
        }

        return "Success";
    }

    private String getDeviceStatus() {
        // Simulate getting device status.
        return "online";
    }
}
```

JavaScript

```
const greengrasssdk = require('aws-greengrass-core-sdk');

const iotClient = new greengrasssdk.IotData();

/**
 * Receives commands from IoT Core and sends telemetry back.
 */
exports.handler = function(event, context) {
    console.log('Received command from IoT Core:', JSON.stringify(event));

    const command = event.command;
    const deviceId = event.device_id || 'device1';

    console.log(`Processing command: ${command}`);

    if (command === 'get_status') {
        const status = 'online';

        const telemetryData = {
            device_id: deviceId,
```

```
            status: status,
            timestamp: Date.now() / 1000
        };

        // Publish telemetry to IoT Core using greengrasssdk.
        const params = {
            topic: `telemetry/${deviceId}`,
            payload: JSON.stringify(telemetryData)
        };

        iotClient.publish(params, (err) => {
            if (err) {
                console.error('Error publishing telemetry:', err);
                context.fail(err);
            } else {
                console.log('Telemetry sent to IoT Core:',
 JSON.stringify(telemetryData));
                context.succeed('Success');
            }
        });
    } else {
        context.succeed('Success');
    }
};
```

C

```
#include <aws/greengrass/greengrasssdk.h>
#include <stdio.h>
#include <string.h>
#include <jansson.h>
#include <time.h>

static aws_greengrass_iot_data_client *iot_client = NULL;

const char* get_device_status(void) {
    // Simulate getting device status.
    return "online";
}

void on_cloud_command(const char *topic, const uint8_t *payload, size_t payload_len,
 void *user_data) {
    // Parse incoming command from IoT Core.
```

```
    char *payload_str = strndup((char *)payload, payload_len);
    json_error_t error;
    json_t *event = json_loads(payload_str, 0, &error);
    free(payload_str);

    if (!event) {
        fprintf(stderr, "Error parsing JSON: %s\n", error.text);
        return;
    }

    // Extract command and device_id.
    json_t *command_obj = json_object_get(event, "command");
    json_t *device_id_obj = json_object_get(event, "device_id");

    const char *command = json_string_value(command_obj);
    const char *device_id = device_id_obj ? json_string_value(device_id_obj) :
"device1";

    printf("Received command from cloud: %s\n", command);

    // Process command.
    if (command && strcmp(command, "get_status") == 0) {
        const char *status = get_device_status();

        // Send telemetry back to IoT Core.
        json_t *telemetry_data = json_object();
        json_object_set_new(telemetry_data, "device_id", json_string(device_id));
        json_object_set_new(telemetry_data, "status", json_string(status));
        json_object_set_new(telemetry_data, "timestamp", json_real(time(NULL)));

        char *telemetry_payload = json_dumps(telemetry_data, JSON_COMPACT);

        // Publish telemetry to IoT Core.
        char telemetry_topic[256];
        snprintf(telemetry_topic, sizeof(telemetry_topic), "telemetry/%s",
device_id);

        aws_greengrass_publish_params params = {
            .topic = telemetry_topic,
            .payload = (uint8_t *)telemetry_payload,
            .payload_len = strlen(telemetry_payload)
        };

        aws_greengrass_iot_data_publish(iot_client, &params);
```

```
        printf("Telemetry sent to cloud: %s\n", telemetry_payload);

        free(telemetry_payload);
        json_decref(telemetry_data);
    }

    json_decref(event);
}

int main(int argc, char *argv[]) {
    // Initialize Greengrass SDK.
    iot_client = aws_greengrass_iot_data_client_new();

    // Subscribe to commands from IoT Core.
    aws_greengrass_subscribe_params subscribe_params = {
        .topic = "commands/device1",
        .callback = on_cloud_command,
        .user_data = NULL
    };

    aws_greengrass_iot_data_subscribe(iot_client, &subscribe_params);

    printf("Device Controller Lambda started\n");
    printf("Subscribed to commands/device1\n");
    printf("Waiting for commands from IoT Core...\n");

    // Keep the Lambda running.
    while (1) {
        sleep(1);
    }

    return 0;
}
```

C++

```
#include <aws/greengrass/greengrasssdk.h>
#include <iostream>
#include <string>
#include <memory>
#include <jansson.h>
#include <ctime>
```

```
#include <unistd.h>

class DeviceController {
private:
    std::unique_ptr<aws_greengrass_iot_data_client,
 decltype(&aws_greengrass_iot_data_client_destroy)> iot_client;

    static void message_callback_wrapper(const char *topic, const uint8_t *payload,
                                         size_t payload_len, void *user_data) {
        auto* controller = static_cast<DeviceController*>(user_data);
        controller->on_cloud_command(topic, payload, payload_len);
    }

    std::string get_device_status() {
        // Simulate getting device status.
        return "online";
    }

public:
    DeviceController()
        : iot_client(aws_greengrass_iot_data_client_new(),
                     aws_greengrass_iot_data_client_destroy) {
        if (!iot_client) {
            throw std::runtime_error("Failed to create Greengrass IoT client");
        }
    }

    void on_cloud_command(const char *topic, const uint8_t *payload, size_t
 payload_len) {
        // Parse incoming command from IoT Core.
        std::string payload_str(reinterpret_cast<const char*>(payload),
 payload_len);

        json_error_t error;
        json_t *event = json_loads(payload_str.c_str(), 0, &error);

        if (!event) {
            std::cerr << "Error parsing JSON: " << error.text << std::endl;
            return;
        }

        // Extract command and device_id.
        json_t *command_obj = json_object_get(event, "command");
        json_t *device_id_obj = json_object_get(event, "device_id");
```

```
        const char *command = json_string_value(command_obj);
        const char *device_id = device_id_obj ? json_string_value(device_id_obj) :
"device1";

        std::cout << "Received command from cloud: " << command << std::endl;

        // Process command.
        if (command && std::string(command) == "get_status") {
            std::string status = get_device_status();

            // Send telemetry back to IoT Core.
            json_t *telemetry_data = json_object();
            json_object_set_new(telemetry_data, "device_id",
json_string(device_id));
            json_object_set_new(telemetry_data, "status",
json_string(status.c_str()));
            json_object_set_new(telemetry_data, "timestamp",
json_real(std::time(nullptr)));

            char *telemetry_payload = json_dumps(telemetry_data, JSON_COMPACT);

            // Publish telemetry to IoT Core.
            std::string telemetry_topic = "telemetry/" + std::string(device_id);

            aws_greengrass_publish_params params = {
                .topic = telemetry_topic.c_str(),
                .payload = reinterpret_cast<uint8_t*>(telemetry_payload),
                .payload_len = strlen(telemetry_payload)
            };

            aws_greengrass_iot_data_publish(iot_client.get(), &params);

            std::cout << "Telemetry sent to cloud: " << telemetry_payload <<
std::endl;

            free(telemetry_payload);
            json_decref(telemetry_data);
        }

        json_decref(event);
    }

    void subscribe_to_topic(const std::string& topic) {
```

```
        aws_greengrass_subscribe_params subscribe_params = {
            .topic = topic.c_str(),
            .callback = message_callback_wrapper,
            .user_data = this
        };

        aws_greengrass_iot_data_subscribe(iot_client.get(), &subscribe_params);

        std::cout << "Device Controller Lambda started" << std::endl;
        std::cout << "Subscribed to " << topic << std::endl;
        std::cout << "Waiting for commands from IoT Core..." << std::endl;
    }

    void run() {
        // Keep the Lambda running.
        while (true) {
            sleep(1);
        }
    }
};

int main(int argc, char *argv[]) {
    try {
        DeviceController controller;
        controller.subscribe_to_topic("commands/device1");
        controller.run();
    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << std::endl;
        return 1;
    }

    return 0;
}
```

Generic component (V2)

To achieve the same functionality in AWS IoT Greengrass V2, create a generic component with the following:

1. Component code

Python

Prerequisites: Before using this component code, install and verify the AWS IoT Device SDK for Python on your Greengrass device:

```
# Install the SDK
pip3 install awsiotsdk

# Verify installation
python3 -c "import awsiot.greengrasscoreipc.clientv2; print('SDK installed
 successfully')"
```

If you encounter dependency conflicts during installation, try installing a specific version of the AWS IoT Device SDK.

If the verification command prints "SDK installed successfully", you're ready to use the component code:

```
from awsiot.greengrasscoreipc.clientv2 import GreengrassCoreIPCClientV2
from awsiot.greengrasscoreipc.model import QOS
import json
import time

ipc_client = GreengrassCoreIPCClientV2()

def on_command(event):
    """
    Receives commands from IoT Core,
    processes them, and sends telemetry back to cloud
    """
    try:
        # Receive command from IoT Core.
        data = json.loads(event.message.payload.decode('utf-8'))
        command = data.get('command')
        device_id = data.get('device_id', 'device1')

        print(f"Received command from cloud: {command}")

        # Process command.
        if command == 'get_status':
            status = get_device_status()
```

```
            # Send telemetry back to IoT Core.
            telemetry_data = {
                'device_id': device_id,
                'status': status,
                'timestamp': time.time()
            }

            ipc_client.publish_to_iot_core(
                topic_name=f'telemetry/{device_id}',
                qos=QOS.AT_LEAST_ONCE,
                payload=json.dumps(telemetry_data).encode('utf-8')
            )

            print(f"Telemetry sent to cloud: {telemetry_data}")

    except Exception as e:
        print(f"Error processing command: {e}")

def get_device_status():
    """Get current device status"""
    # Simulate getting device status.
    return 'online'

def main():
    print("Device Controller component starting...")

    # Subscribe to commands from IoT Core.
    ipc_client.subscribe_to_iot_core(
        topic_name='commands/device1',
        qos=QOS.AT_LEAST_ONCE,
        on_stream_event=on_command
    )

    print("Subscribed to commands/device1 from IoT Core")
    print("Waiting for commands from cloud...")

    # Keep running.
    while True:
        time.sleep(1)

if __name__ == '__main__':
    main()
```

Java

```
import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClientV2;
import software.amazon.awssdk.aws.greengrass.model.QOS;
import software.amazon.awssdk.aws.greengrass.model.SubscribeToIoTCoreRequest;
import software.amazon.awssdk.aws.greengrass.model.IoTCoreMessage;
import software.amazon.awssdk.aws.greengrass.model.PublishToIoTCoreRequest;
import java.nio.charset.StandardCharsets;
import java.util.Optional;
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class DeviceController {
    private static GreengrassCoreIPCClientV2 ipcClient;

    public static void main(String[] args) {
        System.out.println("Device Controller component starting...");

        try (GreengrassCoreIPCClientV2 client =
 GreengrassCoreIPCClientV2.builder().build()) {
            ipcClient = client;

            SubscribeToIoTCoreRequest subscribeRequest = new
 SubscribeToIoTCoreRequest()
                .withTopicName("commands/device1")
                .withQos(QOS.AT_LEAST_ONCE);

            ipcClient.subscribeToIoTCore(
                subscribeRequest,
                DeviceController::onCommand,
                Optional.empty(),
                Optional.empty()
            );

            System.out.println("Subscribed to commands/device1 from IoT Core");
            System.out.println("Waiting for commands from cloud...");

            while (true) {
                Thread.sleep(1000);
            }
        } catch (Exception e) {
            System.err.println("Error: " + e.getMessage());
            e.printStackTrace();
        }
```

```
    }

    public static void onCommand(IoTCoreMessage message) {
        try {
            String payload = new String(message.getMessage().getPayload(),
StandardCharsets.UTF_8);

            // Simple JSON parsing.
            String command = extractJsonValue(payload, "command");
            String deviceId = extractJsonValue(payload, "device_id");
            if (deviceId == null || deviceId.isEmpty()) {
                deviceId = "device1";
            }

            System.out.println("Received command from cloud: " + command);

            if ("get_status".equals(command)) {
                String status = getDeviceStatus();

                // Build JSON manually.
                String telemetryJson = String.format(
                    "{\"device_id\":\"%s\",\"status\":\"%s\",\"timestamp\":%.3f}",
                    deviceId, status, System.currentTimeMillis() / 1000.0
                );
                byte[] telemetryBytes =
telemetryJson.getBytes(StandardCharsets.UTF_8);

                PublishToIoTCoreRequest publishRequest = new
PublishToIoTCoreRequest()
                    .withTopicName("telemetry/" + deviceId)
                    .withQos(QOS.AT_LEAST_ONCE)
                    .withPayload(telemetryBytes);

                ipcClient.publishToIoTCore(publishRequest);

                System.out.println("Telemetry sent to cloud: " + telemetryJson);
            }
        } catch (Exception e) {
            System.err.println("Error processing command: " + e.getMessage());
        }
    }

    private static String extractJsonValue(String json, String key) {
```

```
        Pattern pattern = Pattern.compile("\"" + Pattern.quote(key) + "\"\\s*:\\s*
\"([^\"]+)\"");
        Matcher matcher = pattern.matcher(json);
        return matcher.find() ? matcher.group(1) : null;
    }

    private static String getDeviceStatus() {
        return "online";
    }
}
```

JavaScript

```
const greengrasscoreipc = require('aws-iot-device-sdk-v2').greengrasscoreipc;

class DeviceController {
    constructor() {
        this.ipcClient = null;
    }

    async start() {
        console.log('Device Controller component starting...');

        try {
            this.ipcClient = greengrasscoreipc.createClient();
            await this.ipcClient.connect();

            const subscribeRequest = {
                topicName: 'commands/device1',
                qos: 1
            };

            const streamingOperation =
 this.ipcClient.subscribeToIoTCore(subscribeRequest);

            streamingOperation.on('message', (message) => {
                this.onCommand(message);
            });

            streamingOperation.on('streamError', (error) => {
                console.error('Stream error:', error);
            });
```

```
            streamingOperation.on('ended', () => {
                console.log('Subscription stream ended');
            });

            await streamingOperation.activate();

            console.log('Subscribed to commands/device1 from IoT Core');
            console.log('Waiting for commands from cloud...');

        } catch (error) {
            console.error('Error starting component:', error);
            process.exit(1);
        }
    }

    async onCommand(message) {
        try {
            const payload = message.message.payload.toString('utf-8');
            const data = JSON.parse(payload);

            const command = data.command;
            const deviceId = data.device_id || 'device1';

            console.log(`Received command from cloud: ${command}`);

            if (command === 'get_status') {
                const status = this.getDeviceStatus();

                const telemetryData = {
                    device_id: deviceId,
                    status: status,
                    timestamp: Date.now() / 1000
                };

                const telemetryJson = JSON.stringify(telemetryData);

                const publishRequest = {
                    topicName: `telemetry/${deviceId}`,
                    qos: 1,
                    payload: Buffer.from(telemetryJson, 'utf-8')
                };

                await this.ipcClient.publishToIoTCore(publishRequest);
                console.log(`Telemetry sent to cloud: ${telemetryJson}`);
```

```
            }

        } catch (error) {
            console.error('Error processing command:', error);
        }
    }

    getDeviceStatus() {
        return 'online';
    }
}

// Start the component.
const controller = new DeviceController();
controller.start();
```

C

```
#include <gg/buffer.h>
#include <gg/error.h>
#include <gg/ipc/client.h>
#include <gg/map.h>
#include <gg/object.h>
#include <gg/sdk.h>
#include <unistd.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <pthread.h>

#define COMMAND_TOPIC "commands/device1"
#define TELEMETRY_TOPIC "telemetry/device1"

typedef struct {
    char device_id[64];
    char command[64];
} CommandData;

static pthread_mutex_t command_mutex = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t command_cond = PTHREAD_COND_INITIALIZER;
static CommandData pending_command;
```

```
static bool has_pending_command = false;

const char* get_device_status(void) {
    // Simulate getting device status.
    return "online";
}

static void *telemetry_publisher_thread(void *arg) {
    (void) arg;

    while (true) {
        pthread_mutex_lock(&command_mutex);
        while (!has_pending_command) {
            pthread_cond_wait(&command_cond, &command_mutex);
        }

        CommandData cmd = pending_command;
        has_pending_command = false;
        pthread_mutex_unlock(&command_mutex);

        // Process command.
        if (strcmp(cmd.command, "get_status") == 0) {
            const char *status = get_device_status();

            // Create telemetry JSON string.
            char telemetry_json[512];
            snprintf(telemetry_json, sizeof(telemetry_json),
                     "{\"device_id\":\"%s\",\"status\":\"%s\",\"timestamp\":%ld}",
                     cmd.device_id, status, time(NULL));

            GgBuffer telemetry_buf = {
                .data = (uint8_t *)telemetry_json,
                .len = strlen(telemetry_json)
            };

            // Publish telemetry to IoT Core.
            GgError ret = ggipc_publish_to_iot_core(GG_STR(TELEMETRY_TOPIC),
 telemetry_buf, 0);

            if (ret != GG_ERR_OK) {
                fprintf(stderr, "Failed to publish telemetry to IoT Core\n");
            } else {
                printf("Telemetry sent to cloud: device_id=%s, status=%s\n",
 cmd.device_id, status);
```

```
            }
        }
    }

    return NULL;
}

static void on_cloud_command(
    void *ctx, GgBuffer topic, GgBuffer payload, GgIpcSubscriptionHandle handle
) {
    (void) ctx;
    (void) topic;
    (void) handle;

    printf("Received command from IoT Core: %.*s\n", (int)payload.len,
 payload.data);

    // Parse JSON payload (comes as raw buffer from IoT Core).
    // For simplicity, we'll do basic string parsing.

    // Extract command and device_id from JSON string.
    char payload_str[512];
    snprintf(payload_str, sizeof(payload_str), "%.*s", (int)payload.len,
 payload.data);

    // Simple JSON parsing (looking for "command":"get_status").
    char *command_start = strstr(payload_str, "\"command\"");
    char *device_id_start = strstr(payload_str, "\"device_id\"");

    if (command_start) {
        pthread_mutex_lock(&command_mutex);

        char *cmd_value = strstr(command_start, ":");
        if (cmd_value) {
            cmd_value = strchr(cmd_value, '"');
            if (cmd_value) {
                cmd_value++;
                char *cmd_end = strchr(cmd_value, '"');
                if (cmd_end) {
                    size_t cmd_len = cmd_end - cmd_value;
                    if (cmd_len < sizeof(pending_command.command)) {
                        strncpy(pending_command.command, cmd_value, cmd_len);
                        pending_command.command[cmd_len] = '\0';
                    }
```

```
                }
            }
        }

        // Extract device_id or use default.
        if (device_id_start) {
            char *dev_value = strstr(device_id_start, ":");
            if (dev_value) {
                dev_value = strchr(dev_value, '"');
                if (dev_value) {
                    dev_value++;
                    char *dev_end = strchr(dev_value, '"');
                    if (dev_end) {
                        size_t dev_len = dev_end - dev_value;
                        if (dev_len < sizeof(pending_command.device_id)) {
                            strncpy(pending_command.device_id, dev_value, dev_len);
                            pending_command.device_id[dev_len] = '\0';
                        }
                    }
                }
            }
        } else {
            strcpy(pending_command.device_id, "device1");
        }

        printf("Received command from cloud: %s\n", pending_command.command);

        has_pending_command = true;
        pthread_cond_signal(&command_cond);
        pthread_mutex_unlock(&command_mutex);
    }
}

int main(void) {
    setvbuf(stdout, NULL, _IONBF, 0);
    printf("Device Controller component starting...\n");

    gg_sdk_init();

    GgError ret = ggipc_connect();
    if (ret != GG_ERR_OK) {
        fprintf(stderr, "Failed to connect to Greengrass nucleus\n");
        exit(1);
    }
```

```
    printf("Connected to Greengrass IPC\n");

    // Start telemetry publisher thread.
    pthread_t telemetry_thread;
    if (pthread_create(&telemetry_thread, NULL, telemetry_publisher_thread, NULL) !=
 0) {
        fprintf(stderr, "Failed to create telemetry publisher thread\n");
        exit(1);
    }

    // Subscribe to commands from IoT Core.
    GgIpcSubscriptionHandle handle;
    ret = ggipc_subscribe_to_iot_core(
        GG_STR(COMMAND_TOPIC),
        0,
        &on_cloud_command,
        NULL,
        &handle
    );

    if (ret != GG_ERR_OK) {
        fprintf(stderr, "Failed to subscribe to IoT Core topic\n");
        exit(1);
    }

    printf("Subscribed to %s\n", COMMAND_TOPIC);
    printf("Waiting for commands from IoT Core...\n");

    while (true) {
        sleep(1);
    }

    return 0;
}
```

C++

```
#include <gg/ipc/client.hpp>
#include <gg/buffer.hpp>
#include <gg/object.hpp>
#include <gg/types.hpp>

#include <chrono>
```

```
#include <condition_variable>
#include <ctime>
#include <iostream>
#include <mutex>
#include <string>
#include <string_view>
#include <thread>

constexpr std::string_view COMMAND_TOPIC = "commands/device1";
constexpr std::string_view TELEMETRY_TOPIC = "telemetry/device1";

struct CommandData {
    std::string device_id;
    std::string command;
};

static std::mutex command_mutex;
static std::condition_variable command_cv;
static CommandData pending_command;
static bool has_pending_command = false;

std::string get_device_status() {
    // Simulate getting device status.
    return "online";
}

void telemetry_publisher_thread() {
    auto& client = gg::ipc::Client::get();

    while (true) {
        std::unique_lock<std::mutex> lock(command_mutex);
        command_cv.wait(lock, [] { return has_pending_command; });

        CommandData cmd = pending_command;
        has_pending_command = false;
        lock.unlock();

        // Process command.
        if (cmd.command == "get_status") {
            std::string status = get_device_status();

            // Get current timestamp.
            auto now = std::time(nullptr);
```

```
            // Create telemetry JSON payload.
            std::string telemetry_payload = "{\"device_id\":\"" + cmd.device_id +
                                            "\",\"status\":\"" + status +
                                            "\",\"timestamp\":" + std::to_string(now)
 + "}";

            // Publish telemetry to IoT Core.
            gg::Buffer telemetry_buffer(telemetry_payload);
            auto error = client.publish_to_iot_core(TELEMETRY_TOPIC,
 telemetry_buffer);

            if (error) {
                std::cerr << "Failed to publish telemetry to IoT Core: "
                          << error.message() << std::endl;
            } else {
                std::cout << "Telemetry sent to cloud: device_id=" << cmd.device_id
                          << ", status=" << status << std::endl;
            }
        }
    }
}

class CloudCommandCallback : public gg::ipc::IoTCoreTopicCallback {
    void operator()(
        std::string_view topic,
        gg::Object payload,
        gg::ipc::Subscription& handle
    ) override {
        (void) topic;
        (void) handle;

        // Payload is a Buffer containing JSON string from IoT Core.
        if (payload.index() != GG_TYPE_BUF) {
            std::cerr << "Expected Buffer message\n";
            return;
        }

        // Extract buffer.
        auto buffer = gg::get<std::span<uint8_t>>(payload);
        std::string json_str(reinterpret_cast<const char*>(buffer.data()),
 buffer.size());

        std::cout << "Received command from IoT Core: " << json_str << std::endl;
```

```
        // Simple JSON parsing for demo.
        std::string command;
        std::string device_id = "device1";  // Default

        // Extract command.
        size_t cmd_pos = json_str.find("\"command\":");
        if (cmd_pos != std::string::npos) {
            size_t start = json_str.find("\"", cmd_pos + 10) + 1;
            size_t end = json_str.find("\"", start);
            if (end != std::string::npos) {
                command = json_str.substr(start, end - start);
            }
        }

        // Extract device_id if present.
        size_t dev_pos = json_str.find("\"device_id\":");
        if (dev_pos != std::string::npos) {
            size_t start = json_str.find("\"", dev_pos + 12) + 1;
            size_t end = json_str.find("\"", start);
            if (end != std::string::npos) {
                device_id = json_str.substr(start, end - start);
            }
        }

        if (!command.empty()) {
            std::lock_guard<std::mutex> lock(command_mutex);
            pending_command = {device_id, command};
            has_pending_command = true;
            command_cv.notify_one();

            std::cout << "Received command from cloud: " << command << std::endl;
        }
    }
};

int main() {
    // Disable stdout buffering for real-time logging in systemd/Greengrass.
    std::cout.setf(std::ios::unitbuf);

    std::cout << "Device Controller component starting..." << std::endl;

    auto& client = gg::ipc::Client::get();

    auto error = client.connect();
```

```
    if (error) {
        std::cerr << "Failed to connect to Greengrass nucleus: "
                  << error.message() << std::endl;
        return 1;
    }

    std::cout << "Connected to Greengrass IPC" << std::endl;

    // Start telemetry publisher thread.
    std::thread telemetry_thread(telemetry_publisher_thread);
    telemetry_thread.detach();

    // Subscribe to commands from IoT Core.
    static CloudCommandCallback handler;
    error = client.subscribe_to_iot_core(COMMAND_TOPIC, handler);

    if (error) {
        std::cerr << "Failed to subscribe to IoT Core topic: "
                  << error.message() << std::endl;
        return 1;
    }

    std::cout << "Subscribed to " << COMMAND_TOPIC << std::endl;
    std::cout << "Waiting for commands from IoT Core..." << std::endl;

    // Keep running.
    while (true) {
        using namespace std::chrono_literals;
        std::this_thread::sleep_for(1s);
    }

    return 0;
}
```

2. Build and package the component

Some languages require building or packaging before deployment.

Python

Python does not require compilation. The component can use the .py file directly.

Java

To build an executable JAR with all dependencies bundled:

1. Create a pom.xml file in your project directory:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <!-- Basic project information: organization, component name, and version --
>
    <groupId>com.example</groupId>
    <artifactId>device-controller</artifactId>
    <version>1.0.0</version>

    <properties>
        <!-- Java 11 LTS (Long Term Support) is recommended for Greengrass v2
 components -->
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
    </properties>

    <dependencies>
        <!-- AWS IoT Device SDK for Java - provides IPC client for Greengrass v2
 cloud communication -->
        <dependency>
            <groupId>software.amazon.awssdk.iotdevicesdk</groupId>
            <artifactId>aws-iot-device-sdk</artifactId>
            <version>1.25.1</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <!-- Maven Shade Plugin - creates a standalone JAR with all
 dependencies included for Greengrass deployment -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>3.2.4</version>
```

```
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                        <configuration>
                            <transformers>
                                <!-- Set the main class for the executable JAR
-->
                                <transformer
implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                                    <mainClass>DeviceController</mainClass>
                                </transformer>
                            </transformers>
                            <filters>
                                <!-- Exclude signature files to avoid security
exceptions -->
                                <filter>
                                    <artifact>*:*</artifact>
                                    <excludes>
                                        <exclude>META-INF/*.SF</exclude>
                                        <exclude>META-INF/*.DSA</exclude>
                                        <exclude>META-INF/*.RSA</exclude>
                                    </excludes>
                                </filter>
                            </filters>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

2. Build the JAR:

```
mvn clean package
```

   This creates `target/device-controller-1.0.0.jar` with all dependencies included.

3. Upload the JAR to your S3 bucket for deployment.

JavaScript

To package your Node.js component with all dependencies:

1. Create a `package.json` file:

```
{
  "name": "device-controller",
  "version": "1.0.0",
  "description": "Device controller component for Greengrass v2",
  "main": "DeviceController.js",
  "dependencies": {
    "aws-iot-device-sdk-v2": "^1.21.0"
  },
  "engines": {
    "node": ">=14.0.0"
  }
}
```

2. Install dependencies on your development machine:

```
npm install
```

   This creates a `node_modules` folder containing the AWS AWS IoT Device SDK v2.

3. Package for deployment:

```
zip -r DeviceController.zip DeviceController.js node_modules/ package.json
```

4. Upload the zip file to your S3 bucket for deployment.

> **Note**
>
> The Greengrass device must have Node.js runtime installed (version 14 or later). You don't need to run `npm install` on the Greengrass core device since the component artifact includes all dependencies in the bundled `node_modules` folder.

C

Prerequisites:

To build the SDK and the component, you will need the following build dependencies:

- GCC or Clang
- CMake (at least version 3.22)
- Make or Ninja

Install build dependencies:

On Ubuntu/Debian:

```
sudo apt install build-essential cmake
```

On Amazon Linux:

```
sudo yum install gcc cmake make
```

Create a CMakeLists.txt file for your component:

```
cmake_minimum_required(VERSION 3.10)
project(DeviceController C)

set(CMAKE_C_STANDARD 11)

# Add AWS Greengrass Component SDK
add_subdirectory(aws-greengrass-component-sdk)

# Build your component executable
add_executable(device_controller device_controller.c)
target_link_libraries(device_controller gg-sdk)
```

Build steps:

```
# Clone the AWS Greengrass Component SDK into your project
git clone https://github.com/aws-greengrass/aws-greengrass-component-sdk.git

# Build your component
cmake -B build -D CMAKE_BUILD_TYPE=MinSizeRel
make -C build -j$(nproc)

# The binary 'device_controller' is in ./build/
```

```
# Upload this binary to S3 for deployment
```

C++

Prerequisites:

To build the SDK and the component, you will need the following build dependencies:

- GCC or Clang with C++20 support
- CMake (at least version 3.22)
- Make or Ninja

Install build dependencies:

On Ubuntu/Debian:

```
sudo apt install build-essential cmake
```

On Amazon Linux:

```
sudo yum install gcc-c++ cmake make
```

Create a CMakeLists.txt file for your component:

```
cmake_minimum_required(VERSION 3.10)
project(DeviceController CXX)

set(CMAKE_CXX_STANDARD 20)

# Add SDK as subdirectory
add_subdirectory(aws-greengrass-component-sdk)

# Add C++ SDK subdirectory
add_subdirectory(aws-greengrass-component-sdk/cpp)

add_executable(device_controller device_controller.cpp)
target_link_libraries(device_controller gg-sdk++)
```

Build steps:

```
# Clone the AWS Greengrass Component SDK into your project
```

```
git clone https://github.com/aws-greengrass/aws-greengrass-component-sdk.git

# Build your component
cmake -B build -D CMAKE_BUILD_TYPE=MinSizeRel
make -C build -j$(nproc)

# The binary 'device_controller' will be in ./build/
# Upload this binary to S3 for deployment
```

3. Component recipe

Update the "resources" array with the actual topics your component uses.

Python

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.DeviceController",
  "ComponentVersion": "1.0.0",
  "ComponentType": "aws.greengrass.generic",
  "ComponentDescription": "Receives commands from IoT Core and sends telemetry back
 to cloud",
  "ComponentPublisher": "[Your Company]",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.mqttproxy": {
          "com.example.DeviceController:mqttproxy:1": {
            "policyDescription": "Allows access to subscribe to IoT Core topics",
            "operations": [
              "aws.greengrass#SubscribeToIoTCore"
            ],
            "resources": [
              "commands/device1"
            ]
          },
          "com.example.DeviceController:mqttproxy:2": {
            "policyDescription": "Allows access to publish to IoT Core topics",
            "operations": [
              "aws.greengrass#PublishToIoTCore"
            ],
            "resources": [
              "telemetry/device1"
```

```
            ]
          }
        }
      }
    }
  },
  "ComponentDependencies": {
    "aws.greengrass.TokenExchangeService": {
      "VersionRequirement": ">=2.0.0",
      "DependencyType": "HARD"
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux",
        "runtime": "*"
      },
      "Lifecycle": {
        "run": "python3 -u {artifacts:path}/device_controller.py"
      },
      "Artifacts": [
        {
          "Uri": "s3://YOUR-BUCKET/artifacts/com.example.DeviceController/1.0.0/
device_controller.py"
        }
      ]
    }
  ]
}
```

Java

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.DeviceController",
  "ComponentVersion": "1.0.0",
  "ComponentType": "aws.greengrass.generic",
  "ComponentDescription": "Receives commands from IoT Core and sends telemetry back
 to cloud",
  "ComponentPublisher": "[Your Company]",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
```

```
      "accessControl": {
        "aws.greengrass.ipc.mqttproxy": {
          "com.example.DeviceController:mqttproxy:1": {
            "policyDescription": "Allows access to subscribe to IoT Core topics",
            "operations": [
              "aws.greengrass#SubscribeToIoTCore"
            ],
            "resources": [
              "commands/device1"
            ]
          },
          "com.example.DeviceController:mqttproxy:2": {
            "policyDescription": "Allows access to publish to IoT Core topics",
            "operations": [
              "aws.greengrass#PublishToIoTCore"
            ],
            "resources": [
              "telemetry/device1"
            ]
          }
        }
      }
    },
    "ComponentDependencies": {
      "aws.greengrass.TokenExchangeService": {
        "VersionRequirement": ">=2.0.0",
        "DependencyType": "HARD"
      }
    },
    "Manifests": [
      {
        "Platform": {
          "os": "linux",
          "runtime": "*"
        },
        "Lifecycle": {
          "run": "java -jar {artifacts:path}/DeviceController.jar"
        },
        "Artifacts": [
          {
            "Uri": "s3://YOUR-BUCKET/artifacts/com.example.DeviceController/1.0.0/
DeviceController.jar"
          }
```

```
      ]
    }
  ]
}
```

JavaScript

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.DeviceController",
  "ComponentVersion": "1.0.0",
  "ComponentType": "aws.greengrass.generic",
  "ComponentDescription": "Receives commands from IoT Core and sends telemetry back
 to cloud",
  "ComponentPublisher": "[Your Company]",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.mqttproxy": {
          "com.example.DeviceController:mqttproxy:1": {
            "policyDescription": "Allows access to subscribe to command topics from
 IoT Core",
            "operations": [
              "aws.greengrass#SubscribeToIoTCore"
            ],
            "resources": [
              "commands/device1"
            ]
          },
          "com.example.DeviceController:mqttproxy:2": {
            "policyDescription": "Allows access to publish telemetry to IoT Core",
            "operations": [
              "aws.greengrass#PublishToIoTCore"
            ],
            "resources": [
              "telemetry/*"
            ]
          }
        }
      }
    }
  },
  "Manifests": [
```

```
    {
      "Platform": {
        "os": "linux",
        "runtime": "*"
      },
      "Lifecycle": {
        "run": "cd {artifacts:decompressedPath}/DeviceController && node
 DeviceController.js"
      },
      "Artifacts": [
        {
          "Uri": "s3://YOUR-BUCKET/artifacts/com.example.DeviceController/1.0.0/
DeviceController.zip",
          "Unarchive": "ZIP"
        }
      ]
    }
  ]
}
```

C/C++

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.DeviceController",
  "ComponentVersion": "1.0.0",
  "ComponentType": "aws.greengrass.generic",
  "ComponentDescription": "Receives commands from IoT Core and sends telemetry back
 to cloud",
  "ComponentPublisher": "[Your Company]",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.mqttproxy": {
          "com.example.DeviceController:mqttproxy:1": {
            "policyDescription": "Allows access to subscribe to IoT Core topics",
            "operations": ["aws.greengrass#SubscribeToIoTCore"],
            "resources": ["commands/device1"]
          },
          "com.example.DeviceController:mqttproxy:2": {
            "policyDescription": "Allows access to publish to IoT Core topics",
            "operations": ["aws.greengrass#PublishToIoTCore"],
            "resources": ["telemetry/device1"]
```

```
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux",
        "runtime": "*"
      },
      "Lifecycle": {
        "run": "{artifacts:path}/device_controller"
      },
      "Artifacts": [
        {
          "Uri": "s3://YOUR-BUCKET/artifacts/com.example.DeviceController/1.0.0/
device_controller"
        }
      ]
    }
  ]
}
```

Upgrade Greengrass V1 core devices to Greengrass V2

After you verify that your applications and components work on an AWS IoT Greengrass V2 core device, you can install the AWS IoT Greengrass Core software v2.x on your devices that currently run v1.x, such as production devices. Then, deploy Greengrass V2 components to run your Greengrass applications on the devices.

To upgrade a fleet of devices from V1 to V2, complete these steps for each device to upgrade. You can use thing groups to deploy V2 components to a fleet of core devices.

Tip

We recommend that you create a script to automate the upgrade process for a fleet of devices. If you use AWS Systems Manager to manage your fleet, you can use Systems Manager to run that script on each device to upgrade your fleet from V1 to V2.

You can contact your AWS Enterprise Support representative with questions about how to best automate the upgrade process.

Step 1: Install the AWS IoT Greengrass Core software v2.x

Choose from the following options to install the AWS IoT Greengrass Core software v2.x on a V1 core device:

- **Upgrade in fewer steps**

  To upgrade in fewer steps, you can uninstall the v1.x software before you install the v2.x software.

- **Upgrade with minimal downtime**

  To upgrade with minimal downtime, you can install both versions of the AWS IoT Greengrass Core software at the same time. After you install the AWS IoT Greengrass Core software v2.x and verify that your V2 applications operate correctly, you uninstall the AWS IoT Greengrass Core software v1.x. Before you choose this option, consider the additional RAM required to run both versions of the AWS IoT Greengrass Core software at the same time.

Uninstall AWS IoT Greengrass Core v1.x before you install v2.x

If you want to upgrade sequentially, uninstall the AWS IoT Greengrass Core software v1.x before you install v2.x on your device.

To uninstall the AWS IoT Greengrass Core software v1.x

1. If the AWS IoT Greengrass Core software v1.x is running as a service, you must stop, disable, and remove the service.

   a. Stop the running AWS IoT Greengrass Core software v1.x service.

   ```
   sudo systemctl stop greengrass
   ```

   b. Wait until the service stops. You can use the `list` command to check the status of the service.

```
sudo systemctl list-units --type=service | grep greengrass
```

c. Disable the service.

```
sudo systemctl disable greengrass
```

d. Remove the service.

```
sudo rm /etc/systemd/system/greengrass.service
```

2. If the AWS IoT Greengrass Core software v1.x is not running as a service, use the following command to stop the daemon. Replace *greengrass-root* with the name of your Greengrass root folder. The default location is `/greengrass`.

```
cd /greengrass-root/ggc/core/
sudo ./greengrassd stop
```

3. (Optional) Back up your Greengrass root folder and, if applicable, your custom write folder, to a different folder on your device.

   a. Use the following command to copy the current Greengrass root folder to a different folder, and then remove the root folder.

```
sudo cp -r /greengrass-root /path/to/greengrass-backup
rm -rf /greengrass-root
```

   b. Use the following command to move the write folder to a different folder, and then remove the write folder.

```
sudo cp -r /write-directory /path/to/write-directory-backup
rm -rf /write-directory
```

For Greengrass nucleus: You can use the installation instructions for AWS IoT Greengrass V2 to install the Greengrass nucleus on your device.

For Greengrass nucleus lite: You can use the installation instructions for Greengrass nucleus lite to install Greengrass nucleus lite.

> **Tip**
>
> To reuse a core device's identity when you migrate it from V1 to V2, follow instructions to install the AWS IoT Greengrass Core software with manual provisioning. First remove the V1 core software from the device, and then reuse the V1 core device's AWS IoT thing and certificate, and update the certificate's AWS IoT policies to grant permissions that the v2.x software requires.

Install AWS IoT Greengrass Core software v2.x on a device already running v1.x

If you install the AWS IoT Greengrass Core v2.x software on a device that is already running the AWS IoT Greengrass Core software v1.x, keep the following in mind:

- The AWS IoT thing name for your V2 core device must be unique. Don't use the same thing name as your V1 core device.
- The ports that you use for the AWS IoT Greengrass Core software v2.x must be different from the ports that you use for v1.x.
  - Configure the V1 stream manager to use a port other than 8088. For more information, see Configure stream manager.
  - Configure the V1 MQTT broker to use a port other than 8883. For more information, see Configure the MQTT port for local messaging.
- AWS IoT Greengrass V2 doesn't provide the option to rename the Greengrass system service. If you run Greengrass as a system service, you must do one of the following to avoid conflicting system service names:
  - Rename the Greengrass service for v1.x before you install v2.x.
  - Install the AWS IoT Greengrass Core software v2.x without a system service, and then manually configure the software as a system service with a name other than `greengrass`.

To rename the Greengrass service for v1.x

1. Stop the AWS IoT Greengrass Core software v1.x service.

   ```
   sudo systemctl stop greengrass
   ```

2. Wait for the service to stop. The service can take up to a few minutes to stop. You can use the `list-units` command to check whether the service stopped.

```
sudo systemctl list-units --type=service | grep greengrass
```

3. Disable the service.

```
sudo systemctl disable greengrass
```

4. Rename the service.

```
sudo mv /etc/systemd/system/greengrass.service /etc/systemd/system/greengrass-v1.service
```

5. Reload the service and start it.

```
sudo systemctl daemon-reload
sudo systemctl reset-failed
sudo systemctl enable greengrass-v1
sudo systemctl start greengrass-v1
```

You can then use the installation instructions for AWS IoT Greengrass V2 to install the software on your device.

> **Tip**
>
> To reuse a core device's identity when you migrate it from V1 to V2, follow instructions to install the AWS IoT Greengrass Core software with manual provisioning. First remove the V1 core software from the device, and then reuse the V1 core device's AWS IoT thing and certificate, and update the certificate's AWS IoT policies to grant permissions that the v2.x software requires.

Step 2: Deploy AWS IoT Greengrass V2 components to the core devices

After you install the AWS IoT Greengrass Core software v2.x on your device, deploy components based on which runtime you chose.

For Greengrass nucleus:

Create a deployment that includes the following resources. To deploy components to a fleet of similar devices, create a deployment for a thing group that contains those devices.

- Lambda function components that you created from your V1 Lambda functions. For more information, see Run AWS Lambda functions.
- If you use V1 subscriptions, the legacy subscription router component.
- If you use stream manager, the stream manager component. For more information, see Manage data streams on Greengrass core devices.
- If you use local secrets, the secret manager component.
- If you use V1 connectors, the AWS-provided connector components.
- If you use Docker containers, the Docker application manager component. For more information, see Run a Docker container.
- If you use connected devices, the components for client device support. You must also enable client device support and associate the client devices with your core device. For more information, see Interact with local IoT devices.
- If you use device shadows, the shadow manager component. For more information, see Interact with device shadows.
- If you upload logs from Greengrass core devices to Amazon CloudWatch Logs, the log manager component. For more information, see Monitor AWS IoT Greengrass logs.
- If you integrate with AWS IoT SiteWise, follow instructions to set up the V2 core device as an AWS IoT SiteWise gateway. AWS IoT SiteWise provides an installation script that deploys the AWS IoT SiteWise components for you.
- User-defined components that you developed to implement custom functionality.

For information about creating and revising deployments, see Deploy AWS IoT Greengrass components to devices.

For Greengrass nucleus lite:

Deploy the generic components you created in Step 2 of the migration guide to your Greengrass nucleus lite devices:

1. Create your components in using the component recipes you created

2. Create a deployment targeting your Greengrass nucleus lite devices that includes your generic components
3. Verify that your components are running correctly

Tutorial: Getting started with AWS IoT Greengrass V2

You can complete this getting started tutorial to learn the basic features of AWS IoT Greengrass V2. In this tutorial, you do the following:

1. Install and configure the AWS IoT Greengrass Core software on a Linux device, such as a Raspberry Pi, or a Windows device. This device is a Greengrass core device.
2. Develop a Hello World component on your Greengrass core device. Components are software modules that run on Greengrass core devices.
3. Upload that component to AWS IoT Greengrass V2 in the AWS Cloud.
4. Deploy that component from the AWS Cloud to your Greengrass core device.

> **Note**
>
> This tutorial describes how to set up a development environment and explore the features of AWS IoT Greengrass. For more information about how to set up and configure production devices, see the following:
>
> - Setting up AWS IoT Greengrass core devices
> - Install the AWS IoT Greengrass Core software

You can expect to spend 20 to 30 minutes on this tutorial.

Topics

- Prerequisites
- Sign up for an AWS account
- Step 2: Set up your environment
- Step 3: Install the AWS IoT Greengrass Core software
- Step 4: Develop and test a component on your device
- Step 5: Create your component in the AWS IoT Greengrass service
- Step 6: Deploy your component
- Next steps

Prerequisites

To complete this getting started tutorial, you need the following:

- An AWS account. If you don't have one, see Step 2: Set up your environment.
- The use of an AWS Region that supports AWS IoT Greengrass V2. For the list of supported Regions, see AWS IoT Greengrass V2 endpoints and quotas in the *AWS General Reference*.
- An AWS Identity and Access Management (IAM) user with administrator permissions.
- A device to set up as a Greengrass core device, such as a Raspberry Pi with Raspberry Pi OS (previously called Raspbian), or a Windows 10 device. You must have administrator permissions on this device, or the ability to acquire administrator privileges, such as through `sudo`. This device must have an internet connection.

  You can also choose to use a different device that meets the requirements to install and run the AWS IoT Greengrass Core software.

  If your development computer meets these requirements, you can set it up as your Greengrass core device in this tutorial.
- Python 3.5 or later installed for all users on the device and added to the `PATH` environment variable. On Windows, you must also have the Python Launcher for Windows installed for all users.

  Important

  In Windows, Python doesn't install for all users by default. When you install Python, you must customize the installation to configure it for the AWS IoT Greengrass Core software to run Python scripts. For example, if you use the graphical Python installer, do the following:

  1. Select **Install launcher for all users (recommended)**.
  2. Choose **Customize installation**.
  3. Choose **Next**.
  4. Select **Install for all users**.
  5. Select **Add Python to environment variables**.
  6. Choose **Install**.

For more information, see Using Python on Windows in the *Python 3 documentation*.

- AWS Command Line Interface (AWS CLI) installed and configured with credentials on your development computer and on your device. Make sure you use the same AWS Region to configure the AWS CLI on your development computer and on your device. To use AWS IoT Greengrass V2 with the AWS CLI, you must have one of the following versions or later:
  - Minimum AWS CLI V1 version: v1.18.197
  - Minimum AWS CLI V2 version: v2.1.11

  Tip

  You can run the following command to check the version of the AWS CLI that you have.

  ```
  aws --version
  ```

  For more information, see Installing, updating, and uninstalling the AWS CLI and Configuring the AWS CLI in the *AWS Command Line Interface User Guide*.

  Note

  If you use a 32-bit ARM device, such as a Raspberry Pi with a 32-bit operating system, install AWS CLI V1. AWS CLI V2 isn't available for 32-bit ARM devices. For more information, see Installing, updating, and uninstalling the AWS CLI version 1.

Sign up for an AWS account

To get started with AWS, you need an AWS account. There are two ways to sign up for AWS. If you want complete control over account configuration and where your content is stored, or if you have regulated workloads such as HIPAA or FedRAMP, use Sign up for AWS (advanced). If you are a new user and want to use a login you already own to sign in to AWS and create an AWS environment with preconfigured defaults in select AWS Regions, use Sign up for AWS.

Step 2: Set up your environment

> **Note**
>
> These steps do not apply to nucleus lite.

Follow the steps in this section to set up a Linux or Windows device to use as your AWS IoT Greengrass core device.

Set up a Linux device (Raspberry Pi)

These steps assume that you use a Raspberry Pi with Raspberry Pi OS. If you use a different device or operating system, consult the relevant documentation for your device.

To set up a Raspberry Pi for AWS IoT Greengrass V2

1. Enable SSH on your Raspberry Pi to remotely connect to it. For more information, see SSH (Secure shell) in the *Raspberry Pi Documentation*.
2. Find the IP address of your Raspberry Pi to connect to it with SSH. To do so, you can run the following command on your Raspberry Pi.

   ```
   hostname -I
   ```
3. Connect to your Raspberry Pi with SSH.

   On your development computer, run the following command. Replace *username* with the name of the user to sign in, and replace *pi-ip-address* with the IP address that you found in the previous step.

   ```
   ssh username@pi-ip-address
   ```

   > **Important**
   >
   > If your development computer uses an earlier version of Windows, you might not have the `ssh` command, or you might have `ssh` but can't connect to your Raspberry Pi. To connect to your Raspberry Pi, you can install and configure PuTTY, which is a no-

cost, open source SSH client. Consult the PuTTY documentation to connect to your Raspberry Pi.

4. Install the Java runtime, which AWS IoT Greengrass Core software requires to run. On your Raspberry Pi, use the following commands to install Java 11.

```
sudo apt install default-jdk
```

When the installation completes, run the following command to verify that Java runs on your Raspberry Pi.

```
java -version
```

The command prints the version of Java that runs on the device. The output might look similar to the following example.

```
openjdk version "11.0.9.1" 2020-11-04
OpenJDK Runtime Environment (build 11.0.9.1+1-post-Debian-1deb10u2)
OpenJDK 64-Bit Server VM (build 11.0.9.1+1-post-Debian-1deb10u2, mixed mode)
```

Tip: Set kernel parameters on a Raspberry Pi

If your device is a Raspberry Pi, you can complete the following steps to view and update its Linux kernel parameters:

1. Open the `/boot/cmdline.txt` file. This file specifies Linux kernel parameters to apply when the Raspberry Pi boots.

   For example, on a Linux-based system, you can run the following command to use GNU nano to open the file.

   ```
   sudo nano /boot/cmdline.txt
   ```

2. Verify that the `/boot/cmdline.txt` file contains the following kernel parameters. The `systemd.unified_cgroup_hierarchy=0` parameter specifies to use cgroups v1 instead of cgroups v2.

```
cgroup_enable=memory cgroup_memory=1 systemd.unified_cgroup_hierarchy=0
```

If the `/boot/cmdline.txt` file doesn't contain these parameters, or it contains these parameters with different values, update the file to contain these parameters and values.

3. If you updated the `/boot/cmdline.txt` file, reboot the Raspberry Pi to apply the changes.

```
sudo reboot
```

Set up a Linux device (other)

To set up a Linux device for AWS IoT Greengrass V2

1. Install the Java runtime, which AWS IoT Greengrass Core software requires to run. We recommend that you use Amazon Corretto or OpenJDK long-term support versions. Version 8 or higher is required. The following commands show you how to install OpenJDK on your device.

   - For Debian-based or Ubuntu-based distributions:

     ```
     sudo apt install default-jdk
     ```

   - For Red Hat-based distributions:

     ```
     sudo yum install java-11-openjdk-devel
     ```

   - For Amazon Linux 2:

     ```
     sudo amazon-linux-extras install java-openjdk11
     ```

   - For Amazon Linux 2023:

     ```
     sudo dnf install java-11-amazon-corretto -y
     ```

   When the installation completes, run the following command to verify that Java runs on your Linux device.

```
java -version
```

The command prints the version of Java that runs on the device. For example, on a Debian-based distribution, the output might look similar to the following sample.

```
openjdk version "11.0.9.1" 2020-11-04
OpenJDK Runtime Environment (build 11.0.9.1+1-post-Debian-1deb10u2)
OpenJDK 64-Bit Server VM (build 11.0.9.1+1-post-Debian-1deb10u2, mixed mode)
```

2. (Optional) Create the default system user and group that runs components on the device. You can also choose to let the AWS IoT Greengrass Core software installer create this user and group during installation with the `--component-default-user` installer argument. For more information, see Installer arguments.

```
sudo useradd --system --create-home ggc_user
sudo groupadd --system ggc_group
```

3. Verify that the user that runs the AWS IoT Greengrass Core software (typically `root`), has permission to run `sudo` with any user and any group.

   a. Run the following command to open the `/etc/sudoers` file.

   ```
   sudo visudo
   ```

   b. Verify that the permission for the user looks like the following example.

   ```
   root    ALL=(ALL:ALL) ALL
   ```

4. (Optional) To run containerized Lambda functions, you must enable cgroups v1, and you must enable and mount the *memory* and *devices* cgroups. If you don't plan to run containerized Lambda functions, you can skip this step.

   To enable these cgroups options, boot the device with the following Linux kernel parameters.

   ```
   cgroup_enable=memory cgroup_memory=1 systemd.unified_cgroup_hierarchy=0
   ```

   For information about viewing and setting kernel parameters for your device, see the documentation for your operating system and boot loader. Follow the instructions to permanently set the kernel parameters.

5. Install all other required dependencies on your device as indicated by the list of requirements in Device requirements.

Set up a Windows device

To set up a Windows device for AWS IoT Greengrass V2

1. Install the Java runtime, which AWS IoT Greengrass Core software requires to run. We recommend that you use Amazon Corretto or OpenJDK long-term support versions. Version 8 or higher is required.
2. Check whether Java is available on the PATH system variable, and add it if not. The LocalSystem account runs the AWS IoT Greengrass Core software, so you must add Java to the PATH system variable instead of the PATH user variable for your user. Do the following:
   a. Press the Windows key to open the start menu.
   b. Type **`environment variables`** to search for the system options from the start menu.
   c. In the start menu search results, choose **Edit the system environment variables** to open the **System properties** window.
   d. Choose **Environment variables...** to open the **Environment Variables** window.
   e. Under **System variables**, select **Path**, and then choose **Edit**. In the **Edit environment variable** window, you can view each path on a separate line.
   f. Check if the path to the Java installation's `bin` folder is present. The path might look similar to the following example.

      ```
      C:\\Program Files\\Amazon Corretto\\jdk11.0.13_8\\bin
      ```

   g. If the Java installation's `bin` folder is missing from **Path**, choose **New** to add it, and then choose **OK**.
3. Open the Windows Command Prompt (`cmd.exe`) as an administrator.
4. Create the default user in the LocalSystem account on the Windows device. Replace *`password`* with a secure password.

   ```
   net user /add ggc_user password
   ```

Tip

Depending on your Windows configuration, the user's password might be set to expire at a date in the future. To ensure your Greengrass applications continue to operate, track when the password expires, and update it before it expires. You can also set the user's password to never expire.

- To check when a user and its password expire, run the following command.

```
net user ggc_user | findstr /C:expires
```

- To set a user's password to never expire, run the following command.

```
wmic UserAccount where "Name='ggc_user'" set PasswordExpires=False
```

- If you're using Windows 10 or later where the wmic command is deprecated, run the following PowerShell command.

```
Get-CimInstance -Query "SELECT * from Win32_UserAccount WHERE name =
 'ggc_user'" | Set-CimInstance -Property @{PasswordExpires="False"}
```

5. Download and install the PsExec utility from Microsoft on the device.
6. Use the PsExec utility to store the user name and password for the default user in the Credential Manager instance for the LocalSystem account. Replace *password* with the user's password that you set earlier.

```
psexec -s cmd /c cmdkey /generic:ggc_user /user:ggc_user /pass:password
```

If the **PsExec License Agreement** opens, choose **Accept** to agree to the license and run the command.

Note

On Windows devices, the LocalSystem account runs the Greengrass nucleus, and you must use the PsExec utility to store the default user information in the LocalSystem account. Using the Credential Manager application stores this information in the Windows account of the currently logged on user, instead of the LocalSystem account.

Step 3: Install the AWS IoT Greengrass Core software

> **Tip**
>
> We recommend you try out AWS IoT Greengrass AI Agents Context Pack to quickly set up and experiment with AWS IoT Greengrass. The agent context pack will enable AI agents to set up Greengrass Nucleus and Nucleus Lite, deploy components, and troubleshoot common issues.

Follow the steps in this section to set up your Raspberry Pi as a AWS IoT Greengrass core device that you can use for local development. In this section, you download and run an installer that does the following to configure the AWS IoT Greengrass Core software for your device:

- Installs the Greengrass nucleus component. The nucleus is a mandatory component and is the minimum requirement to run the AWS IoT Greengrass Core software on a device. For more information, see Greengrass nucleus component.
- Registers your device as an AWS IoT thing and downloads a digital certificate that allows your device to connect to AWS. For more information, see Device authentication and authorization for AWS IoT Greengrass.
- Adds the device's AWS IoT thing to a thing group, which is a group or fleet of AWS IoT things. Thing groups enable you to manage fleets of Greengrass core devices. When you deploy software components to your devices, you can choose to deploy to individual devices or to groups of devices. For more information, see Managing devices with AWS IoT in the *AWS IoT Core Developer Guide*.
- Creates the IAM role that allows your Greengrass core device to interact with AWS services. By default, this role allows your device to interact with AWS IoT and send logs to Amazon CloudWatch Logs. For more information, see Authorize core devices to interact with AWS services.
- Installs the AWS IoT Greengrass command line interface (`greengrass-cli`), which you can use to test custom components that you develop on the core device. For more information, see Greengrass Command Line Interface.

Install the AWS IoT Greengrass Core software (console)

1. Sign in to the AWS IoT Greengrass console.

2. Under **Get started with Greengrass**, choose **Set up core device**.
3. Under **Step 1: Register a Greengrass core device**, for **Core device name**, enter the name of the AWS IoT thing for your Greengrass core device. If the thing doesn't exist, the installer creates it.
4. Under **Step 2: Add to a thing group to apply a continuous deployment**, for **Thing group**, choose the AWS IoT thing group to which you want to add your core device.

   - If you select **Enter a new group name**, then in **Thing group name**, enter the name of the new group to create. The installer creates the new group for you.
   - If you select **Select an existing group**, then in **Thing group name**, choose the existing group that you want to use.
   - If you select **No group**, then the installer doesn't add the core device to a thing group.
5. Under **Step 3: Install the Greengrass Core software**, complete the following steps.

   Nucleus classic

   1. Choose **Nucleus classic** as your core device's software runtime.
   2. Choose your core device's operating system: **Linux** or **Windows**.
   3. Provide your AWS credentials to the device so that the installer can provision the AWS IoT and IAM resources for your core device. To increase security, we recommend that you get temporary credentials for an IAM role that allows only the minimum permissions necessary to provision. For more information, see Minimal IAM policy for installer to provision resources.

      > **Note**
      >
      > The installer doesn't save or store your credentials.

      On your device, do one of the following to retrieve credentials and make them available to the AWS IoT Greengrass Core software installer:

      - (Recommended) Use temporary credentials from AWS IAM Identity Center

        a. Provide the access key ID, secret access key, and session token from the IAM Identity Center. For more information, see **Manual credential refresh** in Getting and refreshing temporary credentials in the *IAM Identity Center user guide*.

b. Run the following commands to provide the credentials to the AWS IoT Greengrass Core software.

Linux or Unix

```
export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/
bPxRfiCYEXAMPLEKEY
export AWS_SESSION_TOKEN=AQoDYXdzEJr1K...o5OytwEXAMPLE=
```

Windows Command Prompt (CMD)

```
set AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
set AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/
bPxRfiCYEXAMPLEKEY
set AWS_SESSION_TOKEN=AQoDYXdzEJr1K...o5OytwEXAMPLE=
```

PowerShell

```
$env:AWS_ACCESS_KEY_ID="AKIAIOSFODNN7EXAMPLE"
$env:AWS_SECRET_ACCESS_KEY="wJalrXUtnFEMI/K7MDENG/
bPxRfiCYEXAMPLEKEY"
$env:AWS_SESSION_TOKEN="AQoDYXdzEJr1K...o5OytwEXAMPLE="
```

- Use temporary security credentials from an IAM role:

  a. Provide the access key ID, secret access key, and session token from an IAM role that you assume. For more information about how to retrieve these credentials, see Requesting temporary security credentials in the *IAM User Guide*.

  b. Run the following commands to provide the credentials to the AWS IoT Greengrass Core software.

  Linux or Unix

```
export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/
bPxRfiCYEXAMPLEKEY
export AWS_SESSION_TOKEN=AQoDYXdzEJr1K...o5OytwEXAMPLE=
```

Windows Command Prompt (CMD)

```
set AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
set AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/
bPxRfiCYEXAMPLEKEY
set AWS_SESSION_TOKEN=AQoDYXdzEJr1K...o5OytwEXAMPLE=
```

PowerShell

```
$env:AWS_ACCESS_KEY_ID="AKIAIOSFODNN7EXAMPLE"
$env:AWS_SECRET_ACCESS_KEY="wJalrXUtnFEMI/K7MDENG/
bPxRfiCYEXAMPLEKEY"
$env:AWS_SESSION_TOKEN="AQoDYXdzEJr1K...o5OytwEXAMPLE="
```

- Use long-term credentials from an IAM user:

  a. Provide the access key ID and secret access key for your IAM user. You can create an IAM user for provisioning that you later delete. For the IAM policy to give the user, see Minimal IAM policy for installer to provision resources. For more information about how to retrieve long-term credentials, see Managing access keys for IAM users in the *IAM User Guide*.

  b. Run the following commands to provide the credentials to the AWS IoT Greengrass Core software.

  Linux or Unix

  ```
  export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
  export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/
  bPxRfiCYEXAMPLEKEY
  ```

  Windows Command Prompt (CMD)

  ```
  set AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
  set AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/
  bPxRfiCYEXAMPLEKEY
  ```

  PowerShell

  ```
  $env:AWS_ACCESS_KEY_ID="AKIAIOSFODNN7EXAMPLE"
  ```

```
$env:AWS_SECRET_ACCESS_KEY="wJalrXUtnFEMI/K7MDENG/
bPxRfiCYEXAMPLEKEY"
```

c. (Optional) If you created an IAM user to provision your Greengrass device, delete the user.

d. (Optional) If you used the access key ID and secret access key from an existing IAM user, update the keys for the user so that they are no longer valid. For more information, see Updating access keys in the *AWS Identity and Access Management user guide*.

4. Under **Run the installer**, complete the following steps.

   a. Under **Download the installer**, choose **Copy** and run the copied command on your core device. This command downloads the latest version of the AWS IoT Greengrass Core software and unzips it on your device.

   b. Under **Run the installer**, choose **Copy**, and run the copied command on your core device. This command uses the AWS IoT thing and thing group names that you specified earlier to run the AWS IoT Greengrass Core software installer and set up AWS resources for your core device.

      This command also does the following:

      - Set up the AWS IoT Greengrass Core software as a system service that runs at boot. On Linux devices, this requires the Systemd init system.

        > **Important**
        >
        > On Windows core devices, you must set up the AWS IoT Greengrass Core software as a system service.

      - Deploy the AWS IoT Greengrass CLI component, which is a command-line tool that enables you to develop custom Greengrass components on the core device.
      - Specify to use the `ggc_user` system user to run software components on the core device. On Linux devices, this command also specifies to use the `ggc_group` system group, and the installer creates the system user and group for you.

When you run this command, you should see the following messages to indicate that the installer succeeded.

```
Successfully configured Nucleus with provisioned resource details!
Configured Nucleus to deploy aws.greengrass.Cli component
Successfully set up Nucleus as a system service
```

> **Note**
>
> If you have a Linux device and it doesn't have systemd, the installer won't set up the software as a system service, and you won't see the success message for setting up the nucleus as a system service.

Nucleus lite

1. Choose **Nucleus lite** as your core device's software runtime.
2. Select your device set up method to provision your device to a Greengrass core device.

Option 1: Set up a device with package download (approximately 1MB)

1. Create an AWS IoT thing and the role for Greengrass.
2. Download the zip file that contains AWS IoT resources that your device needs to connect to AWS IoT:
   - A certificate and private key generated using AWS IoT's certificate authority.
   - A schema file to initiate Greengrass installation for your device.
3. Download the package that will install the latest Greengrass Nucleus lite runtime to your Raspberry Pi.
4. Provision your device to become an AWS IoT Greengrass Core device and connect it to AWS IoT:

   a. a. Transfer the Greengrass package and connection kit to your device using a USB thumb drive, SCP/FTP, or SD cards.

   b. b. Unzip the greengrass-package.zip file in the /GreengrassInstaller directory on the device.
   c. c. Unzip the connection kit zip file in the /directory on the device.
   d. d. Run the provided command on the device to install AWS IoT Greengrass
5. Then, choose **View core devices**.

Option 2: Set up a device with a pre-configured whole disk sample image download (approximately 100MB)

1. Create an AWS IoT thing and the role for Greengrass.
2. Download the zip file that contains AWS IoT resources that your device needs to connect to AWS IoT:
   - A certificate and private key generated using AWS IoT's certificate authority.
   - A schema file to initiate Greengrass installation for your device.
3. Download the pre-configured whole disk sample image that contains Greengrass and the operating system.
   a. To transfer the connection kit and flash the image onto your device, follow the readme file downloaded with the image.
   b. To start Greengrass installation, turn on and boot the device from the flashed image
4. Then, choose **View core devices**.

Option 3: Set up a device with your own custom build

1. Create an AWS IoT thing and the role for Greengrass.
2. Download the zip file that contains AWS IoT resources that your device needs to connect to AWS IoT:
   - A certificate and private key generated using AWS IoT's certificate authority.
   - A schema file to initiate Greengrass installation for your device.
3. To customize and build your own image using Yocto from source code, and then use the connection kit to install nucleus lite, follow the instructions on GitHub.

- Then, choose **View core devices**.

Install the AWS IoT Greengrass Core software (CLI)

> **Note**
>
> These steps do not apply to nucleus lite.

To install and configure the AWS IoT Greengrass Core software

1. On your Greengrass core device, run the following command to switch to the home directory.

   Linux or Unix

   ```
   cd ~
   ```

   Windows Command Prompt (CMD)

   ```
   cd %USERPROFILE%
   ```

   PowerShell

   ```
   cd ~
   ```

2. On your core device, download the AWS IoT Greengrass Core software to a file named `greengrass-nucleus-latest.zip`.

   Linux or Unix

   ```
   curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-
   latest.zip > greengrass-nucleus-latest.zip
   ```

   Windows Command Prompt (CMD)

   ```
   curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-
   latest.zip > greengrass-nucleus-latest.zip
   ```

PowerShell

```
iwr -Uri https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-
latest.zip -OutFile greengrass-nucleus-latest.zip
```

By downloading this software, you agree to the Greengrass Core Software License Agreement.

3. Unzip the AWS IoT Greengrass Core software to a folder on your device. Replace *GreengrassInstaller* with the folder that you want to use.

   Linux or Unix

   ```
   unzip greengrass-nucleus-latest.zip -d GreengrassInstaller && rm greengrass-
   nucleus-latest.zip
   ```

   Windows Command Prompt (CMD)

   ```
   mkdir GreengrassInstaller && tar -xf greengrass-nucleus-latest.zip -
   C GreengrassInstaller && del greengrass-nucleus-latest.zip
   ```

   PowerShell

   ```
   Expand-Archive -Path greengrass-nucleus-latest.zip -DestinationPath .\
   \GreengrassInstaller
   rm greengrass-nucleus-latest.zip
   ```

4. Run the following command to launch the AWS IoT Greengrass Core software installer. This command does the following:

   - Create the AWS resources that the core device requires to operate.
   - Set up the AWS IoT Greengrass Core software as a system service that runs at boot. On Linux devices, this requires the Systemd init system.

   > **Important**
   >
   > On Windows core devices, you must set up the AWS IoT Greengrass Core software as a system service.

- Deploy the AWS IoT Greengrass CLI component, which is a command-line tool that enables you to develop custom Greengrass components on the core device.
- Specify to use the `ggc_user` system user to run software components on the core device. On Linux devices, this command also specifies to use the `ggc_group` system group, and the installer creates the system user and group for you.

Replace argument values in your command as follows.

a. *`/greengrass/v2`* or *`C:\greengrass\v2`*: The path to the root folder to use to install the AWS IoT Greengrass Core software.

b. *`GreengrassInstaller`*. The path to the folder where you unpacked the AWS IoT Greengrass Core software installer.

c. *`region`*. The AWS Region in which to find or create resources.

d. *`MyGreengrassCore`*. The name of the AWS IoT thing for your Greengrass core device. If the thing doesn't exist, the installer creates it. The installer downloads the certificates to authenticate as the AWS IoT thing. For more information, see Device authentication and authorization for AWS IoT Greengrass.

> **Note**
>
> The thing name can't contain colon (`:`) characters.

e. *`MyGreengrassCoreGroup`*. The name of AWS IoT thing group for your Greengrass core device. If the thing group doesn't exist, the installer creates it and adds the thing to it. If the thing group exists and has an active deployment, the core device downloads and runs the software that the deployment specifies.

> **Note**
>
> The thing group name can't contain colon (`:`) characters.

f. *`GreengrassV2IoTThingPolicy`*. The name of the AWS IoT policy that allows the Greengrass core devices to communicate with AWS IoT and AWS IoT Greengrass. If the AWS IoT policy doesn't exist, the installer creates a permissive AWS IoT policy with this name. You can restrict this policy's permissions for you use case. For more information, see Minimal AWS IoT policy for AWS IoT Greengrass V2 core devices.

g. *GreengrassV2TokenExchangeRole*. The name of the IAM role that allows the Greengrass core device to get temporary AWS credentials. If the role doesn't exist, the installer creates it and creates and attaches a policy named *GreengrassV2TokenExchangeRole*Access. For more information, see Authorize core devices to interact with AWS services.

h. *GreengrassCoreTokenExchangeRoleAlias*. The alias to the IAM role that allows the Greengrass core device to get temporary credentials later. If the role alias doesn't exist, the installer creates it and points it to the IAM role that you specify. For more information, see Authorize core devices to interact with AWS services.

Linux or Unix

```
sudo -E java -Droot="/greengrass/v2" -Dlog.store=FILE \
  -jar ./GreengrassInstaller/lib/Greengrass.jar \
  --aws-region region \
  --thing-name MyGreengrassCore \
  --thing-group-name MyGreengrassCoreGroup \
  --thing-policy-name GreengrassV2IoTThingPolicy \
  --tes-role-name GreengrassV2TokenExchangeRole \
  --tes-role-alias-name GreengrassCoreTokenExchangeRoleAlias \
  --component-default-user ggc_user:ggc_group \
  --provision true \
  --setup-system-service true \
  --deploy-dev-tools true
```

Windows Command Prompt (CMD)

```
java -Droot="C:\greengrass\v2" "-Dlog.store=FILE" ^
  -jar ./GreengrassInstaller/lib/Greengrass.jar ^
  --aws-region region ^
  --thing-name MyGreengrassCore ^
  --thing-group-name MyGreengrassCoreGroup ^
  --thing-policy-name GreengrassV2IoTThingPolicy ^
  --tes-role-name GreengrassV2TokenExchangeRole ^
  --tes-role-alias-name GreengrassCoreTokenExchangeRoleAlias ^
  --component-default-user ggc_user ^
  --provision true ^
  --setup-system-service true ^
  --deploy-dev-tools true
```

PowerShell

```
java -Droot="C:\greengrass\v2" "-Dlog.store=FILE" `
  -jar ./GreengrassInstaller/lib/Greengrass.jar `
  --aws-region region `
  --thing-name MyGreengrassCore `
  --thing-group-name MyGreengrassCoreGroup `
  --thing-policy-name GreengrassV2IoTThingPolicy `
  --tes-role-name GreengrassV2TokenExchangeRole `
  --tes-role-alias-name GreengrassCoreTokenExchangeRoleAlias `
  --component-default-user ggc_user `
  --provision true `
  --setup-system-service true `
  --deploy-dev-tools true
```

Note

If you are running AWS IoT Greengrass on a device with limited memory, you can control the amount of memory that AWS IoT Greengrass Core software uses. To control memory allocation, you can set JVM heap size options in the `jvmOptions` configuration parameter in your nucleus component. For more information, see Control memory allocation with JVM options.

When you run this command, you should see the following messages to indicate that the installer succeeded.

```
Successfully configured Nucleus with provisioned resource details!
Configured Nucleus to deploy aws.greengrass.Cli component
Successfully set up Nucleus as a system service
```

Note

If you have a Linux device and it doesn't have systemd, the installer won't set up the software as a system service, and you won't see the success message for setting up the nucleus as a system service.

(Optional) Run the Greengrass software (Linux)

> **Note**
>
> These steps do not apply to nucleus lite.

If you installed the software as a system service, the installer runs the software for you. Otherwise, you must run the software. To see if the installer set up the software as a system service, look for the following line in the installer output.

```
Successfully set up Nucleus as a system service
```

If you don't see this message, do the following to run the software:

1. Run the following command to run the software.

   ```
   sudo /greengrass/v2/alts/current/distro/bin/loader
   ```

   The software prints the following message if it launches successfully.

   ```
   Launched Nucleus successfully.
   ```

2. You must leave the current command shell open to keep the AWS IoT Greengrass Core software running. If you use SSH to connect to the core device, run the following command on your development computer to open a second SSH session that you can use to run additional commands on the core device. Replace *username* with the name of the user to sign in, and replace *pi-ip-address* with the IP address of the device.

   ```
   ssh username@pi-ip-address
   ```

For more information about how to interact with the Greengrass system service, see Configure the Greengrass nucleus as a system service.

Verify the Greengrass CLI installation on the device

> **Note**
>
> These steps do not apply to nucleus lite.

The Greengrass CLI can take up to a minute to deploy. Run the following command to check the status of the deployment. Replace *MyGreengrassCore* with the name of your core device.

```
aws greengrassv2 list-effective-deployments --core-device-thing-name MyGreengrassCore
```

The `coreDeviceExecutionStatus` indicates the status of the deployment to the core device. When the status is SUCCEEDED, run the following command to verify that the Greengrass CLI is installed and runs. Replace */greengrass/v2* with the path to the root folder.

Linux or Unix

```
/greengrass/v2/bin/greengrass-cli help
```

Windows Command Prompt (CMD)

```
C:\greengrass\v2\bin\greengrass-cli help
```

PowerShell

```
C:\greengrass\v2\bin\greengrass-cli help
```

The command outputs help information for the Greengrass CLI. If the `greengrass-cli` isn't found, the deployment might have failed to install the Greengrass CLI. For more information, see Troubleshooting AWS IoT Greengrass V2.

You can also run the following command to manually deploy the AWS IoT Greengrass CLI to your device.

- Replace *region* with the AWS Region that you use. Make sure that you use the same AWS Region that you used to configure the AWS CLI on your device.

- Replace *account-id* with your AWS account ID.
- Replace *MyGreengrassCore* with the name of your core device.

Linux, macOS, or Unix

```
aws greengrassv2 create-deployment \
  --target-arn "arn:aws:iot:region:account-id:thing/MyGreengrassCore" \
  --components '{
    "aws.greengrass.Cli": {
      "componentVersion": "2.18.4"
    }
  }'
```

Windows Command Prompt (CMD)

```
aws greengrassv2 create-deployment ^
  --target-arn "arn:aws:iot:region:account-id:thing/MyGreengrassCore" ^
  --components "{\"aws.greengrass.Cli\":{\"componentVersion\":\"2.18.4\"}}"
```

PowerShell

```
aws greengrassv2 create-deployment `
  --target-arn "arn:aws:iot:region:account-id:thing/MyGreengrassCore" `
  --components '{\"aws.greengrass.Cli\":{\"componentVersion\":\"2.18.4\"}}'
```

Tip

You can add */greengrass/v2*/bin (Linux) or *C:\greengrass\v2*\bin (Windows) to your PATH environment variable to run `greengrass-cli` without its absolute path.

The AWS IoT Greengrass Core software and local development tools run on your device. Next, you can develop a Hello World AWS IoT Greengrass component on your device.

Step 4: Develop and test a component on your device

A component is a software module that runs on AWS IoT Greengrass core devices. Components enable you to create and manage complex applications as discrete building blocks that you can

reuse from one Greengrass core device to another. Every component is composed of a *recipe* and *artifacts*.

- **Recipes**

  Every component contains a recipe file, which defines its metadata. The recipe also specifies the component's configuration parameters, component dependencies, lifecycle, and platform compatibility. The component lifecycle defines the commands that install, run, and shut down the component. For more information, see AWS IoT Greengrass component recipe reference.

  You can define recipes in JSON or YAML format.

- **Artifacts**

  Components can have any number of artifacts, which are component binaries. Artifacts can include scripts, compiled code, static resources, and any other files that a component consumes. Components can also consume artifacts from component dependencies.

With AWS IoT Greengrass, you can use the Greengrass CLI to develop and test components locally on a Greengrass core device without interaction with the AWS Cloud. When you complete your local component, you can use the component recipe and artifacts to create that component in the AWS IoT Greengrass service in the AWS Cloud, and then deploy it to all of your Greengrass core devices. For more information about components, see Develop AWS IoT Greengrass components.

In this section, you learn how to create and run a basic Hello World component locally on your core device.

To develop a Hello World component on your device

1. Create a folder for your components with subfolders for recipes and artifacts. Run the following commands on your Greengrass core device to create these folders and change to the component folder. Replace *~/greengrassv2* or *%USERPROFILE%\greengrassv2* with the path to the folder to use for local development.

   Linux or Unix

   ```
   mkdir -p ~/greengrassv2/{recipes,artifacts}
   cd ~/greengrassv2
   ```

Windows Command Prompt (CMD)

```
mkdir %USERPROFILE%\greengrassv2\\recipes, %USERPROFILE%\greengrassv2\\artifacts
cd %USERPROFILE%\greengrassv2
```

PowerShell

```
mkdir ~/greengrassv2/recipes, ~/greengrassv2/artifacts
cd ~/greengrassv2
```

2. Use a text editor to create a recipe file that defines your component's metadata, parameters, dependencies, lifecycle, and platform capability. Include the component version in the recipe file name so that you can identify which recipe reflects which component version. You can choose YAML or JSON format for your recipe.

   For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

   JSON

   ```
   nano recipes/com.example.HelloWorld-1.0.0.json
   ```

   YAML

   ```
   nano recipes/com.example.HelloWorld-1.0.0.yaml
   ```

   > **Note**
   >
   > AWS IoT Greengrass uses semantic versions for components. Semantic versions follow a *major.minor.patch* number system. For example, version `1.0.0` represents the first major release for a component. For more information, see the semantic version specification.

3. Paste the following recipe into the file.

   JSON

   ```
   {
   ```

```
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.HelloWorld",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "My first AWS IoT Greengrass component.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "Message": "world"
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "run": "python3 -u {artifacts:path}/hello_world.py {configuration:/
Message}"
      }
    },
    {
      "Platform": {
        "os": "windows"
      },
      "Lifecycle": {
        "run": "py -3 -u {artifacts:path}/hello_world.py {configuration:/
Message}"
      }
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.HelloWorld
ComponentVersion: '1.0.0'
ComponentDescription: My first AWS IoT Greengrass component.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    Message: world
```

```
Manifests:
  - Platform:
      os: linux
    Lifecycle:
      run: |
        python3 -u {artifacts:path}/hello_world.py "{configuration:/Message}"
  - Platform:
      os: windows
    Lifecycle:
      run: |
        py -3 -u {artifacts:path}/hello_world.py "{configuration:/Message}"
```

This recipe's `ComponentConfiguration` section defines a parameter, `Message`, that defaults to `world`. The `Manifests` section defines a *manifest*, which is a set of lifecycle instructions and artifacts for a platform. You can define multiple manifests to specify different install instructions for various platforms, for example. In the manifest, the `Lifecycle` section instructs the Greengrass core device to run the Hello World script with the `Message` parameter value as an argument.

4. Run the following command to create a folder for the component artifacts.

   Linux or Unix

   ```
   mkdir -p artifacts/com.example.HelloWorld/1.0.0
   ```

   Windows Command Prompt (CMD)

   ```
   mkdir artifacts\com.example.HelloWorld\1.0.0
   ```

   PowerShell

   ```
   mkdir artifacts\com.example.HelloWorld\1.0.0
   ```

   Important

   You must use the following format for the artifact folder path. Include the component name and version that you specify in the recipe.

```
artifacts/componentName/componentVersion/
```

5. Use a text editor to create a Python script artifact file for your Hello World component.

   For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

   ```
   nano artifacts/com.example.HelloWorld/1.0.0/hello_world.py
   ```

   Copy and paste the following Python script into the file.

   ```
   import sys

   message = "Hello, %s!" % sys.argv[1]

   # Print the message to stdout, which Greengrass saves in a log file.
   print(message)
   ```

6. Use the local AWS IoT Greengrass CLI to manage components on your Greengrass core device.

   Run the following command to deploy the component to the AWS IoT Greengrass core. Replace */greengrass/v2* or *C:\greengrass\v2* with your AWS IoT Greengrass V2 root folder, and replace *~/greengrassv2* or *%USERPROFILE%\greengrassv2* with your component development folder.

   Linux or Unix

   ```
   sudo /greengrass/v2/bin/greengrass-cli deployment create \
     --recipeDir ~/greengrassv2/recipes \
     --artifactDir ~/greengrassv2/artifacts \
     --merge "com.example.HelloWorld=1.0.0"
   ```

   Windows Command Prompt (CMD)

   ```
   C:\greengrass\v2\bin\greengrass-cli deployment create ^
     --recipeDir %USERPROFILE%\greengrassv2\recipes ^
     --artifactDir %USERPROFILE%\greengrassv2\artifacts ^
     --merge "com.example.HelloWorld=1.0.0"
   ```

PowerShell

```
C:\greengrass\v2\bin\greengrass-cli deployment create `
  --recipeDir ~/greengrassv2/recipes `
  --artifactDir ~/greengrassv2/artifacts `
  --merge "com.example.HelloWorld=1.0.0"
```

This command adds the component that uses the recipe in `recipes` and the Python script in `artifacts`. The `--merge` option adds or updates the component and version that you specify.

7. The AWS IoT Greengrass Core software saves stdout from component process to log files in the `logs` folder. Run the following command to verify that the Hello World component runs and prints messages.

   Linux or Unix

   ```
   sudo tail -f /greengrass/v2/logs/com.example.HelloWorld.log
   ```

   Windows Command Prompt (CMD)

   ```
   type C:\greengrass\v2\logs\com.example.HelloWorld.log
   ```

   The `type` command writes the file's contents to the terminal. Run this command multiple times to observe changes in the file.

   PowerShell

   ```
   gc C:\greengrass\v2\logs\com.example.HelloWorld.log -Tail 10 -Wait
   ```

   You should see messages similar to the following example.

   ```
   Hello, world!
   ```

Note

If the file doesn't exist, the local deployment may not be complete yet. If the file doesn't exist within 15 seconds, the deployment likely failed. This can occur if your recipe isn't valid, for example. Run the following command to view the AWS IoT Greengrass core log file. This file includes logs from the Greengrass core device's deployment service.

Linux or Unix

```
sudo tail -f /greengrass/v2/logs/greengrass.log
```

Windows Command Prompt (CMD)

```
type C:\greengrass\v2\logs\greengrass.log
```

The `type` command writes the file's contents to the terminal. Run this command multiple times to observe changes in the file.

PowerShell

```
gc C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
```

8. Modify the local component to iterate and test your code. Open `hello_world.py` in a text editor, and add the following code at line 4 to edit the message that the AWS IoT Greengrass core logs.

```
message += " Greetings from your first Greengrass component."
```

The `hello_world.py` script should now have the following contents.

```
import sys

message = "Hello, %s!" % sys.argv[1]
message += " Greetings from your first Greengrass component."

# Print the message to stdout, which Greengrass saves in a log file.
print(message)
```

9. Run the following command to update the component with your changes.

   Linux or Unix

   ```
   sudo /greengrass/v2/bin/greengrass-cli deployment create \
     --recipeDir ~/greengrassv2/recipes \
     --artifactDir ~/greengrassv2/artifacts \
     --merge "com.example.HelloWorld=1.0.0"
   ```

   Windows Command Prompt (CMD)

   ```
   C:\greengrass\v2\bin\greengrass-cli deployment create ^
     --recipeDir %USERPROFILE%\greengrassv2\recipes ^
     --artifactDir %USERPROFILE%\greengrassv2\artifacts ^
     --merge "com.example.HelloWorld=1.0.0"
   ```

   PowerShell

   ```
   C:\greengrass\v2\bin\greengrass-cli deployment create `
     --recipeDir ~/greengrassv2/recipes `
     --artifactDir ~/greengrassv2/artifacts `
     --merge "com.example.HelloWorld=1.0.0"
   ```

   This command updates the `com.example.HelloWorld` component with the latest Hello World artifact.

10. Run the following command to restart the component. When you restart a component, the core device uses the latest changes.

    Linux or Unix

    ```
    sudo /greengrass/v2/bin/greengrass-cli component restart \
      --names "com.example.HelloWorld"
    ```

    Windows Command Prompt (CMD)

    ```
    C:\greengrass\v2\bin\greengrass-cli component restart ^
      --names "com.example.HelloWorld"
    ```

PowerShell

```
C:\greengrass\v2\bin\greengrass-cli component restart `
  --names "com.example.HelloWorld"
```

11. Check the log again to verify that the Hello World component prints the new message.

    Linux or Unix

    ```
    sudo tail -f /greengrass/v2/logs/com.example.HelloWorld.log
    ```

    Windows Command Prompt (CMD)

    ```
    type C:\greengrass\v2\logs\com.example.HelloWorld.log
    ```

    The `type` command writes the file's contents to the terminal. Run this command multiple times to observe changes in the file.

    PowerShell

    ```
    gc C:\greengrass\v2\logs\com.example.HelloWorld.log -Tail 10 -Wait
    ```

    You should see messages similar to the following example.

    ```
    Hello, world! Greetings from your first Greengrass component.
    ```

12. You can update the component's configuration parameters to test different configurations. When you deploy a component, you can specify a *configuration update*, which defines how to modify the component's configuration on the core device. You can specify which configuration values to reset to default values and the new configuration values to merge onto the core device. For more information, see Update component configurations.

    Do the following:

    a. Use a text editor to create a file called `hello-world-config-update.json` to contain the configuration update

       For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

```
nano hello-world-config-update.json
```

b. Copy and paste the following JSON object into the file. This JSON object defines a configuration update that merges the value `friend` to the `Message` parameter to update its value. This configuration update doesn't specify any values to reset. You don't need to reset the `Message` parameter because the merge update replaces the existing value.

```
{
  "com.example.HelloWorld": {
    "MERGE": {
      "Message": "friend"
    }
  }
}
```

c. Run the following command to deploy the configuration update to the Hello World component.

Linux or Unix

```
sudo /greengrass/v2/bin/greengrass-cli deployment create \
  --merge "com.example.HelloWorld=1.0.0" \
  --update-config hello-world-config-update.json
```

Windows Command Prompt (CMD)

```
C:\greengrass\v2\bin\greengrass-cli deployment create ^
  --merge "com.example.HelloWorld=1.0.0" ^
  --update-config hello-world-config-update.json
```

PowerShell

```
C:\greengrass\v2\bin\greengrass-cli deployment create `
  --merge "com.example.HelloWorld=1.0.0" `
  --update-config hello-world-config-update.json
```

d. Check the log again to verify that the Hello World component outputs the new message.

Linux or Unix

```
sudo tail -f /greengrass/v2/logs/com.example.HelloWorld.log
```

Windows Command Prompt (CMD)

```
type C:\greengrass\v2\logs\com.example.HelloWorld.log
```

The `type` command writes the file's contents to the terminal. Run this command multiple times to observe changes in the file.

PowerShell

```
gc C:\greengrass\v2\logs\com.example.HelloWorld.log -Tail 10 -Wait
```

You should see messages similar to the following example.

```
Hello, friend! Greetings from your first Greengrass component.
```

13. After you finish testing your component, remove it from your core device. Run the following command.

    Linux or Unix

    ```
    sudo /greengrass/v2/bin/greengrass-cli deployment create --
    remove="com.example.HelloWorld"
    ```

    Windows Command Prompt (CMD)

    ```
    C:\greengrass\v2\bin\greengrass-cli deployment create --
    remove="com.example.HelloWorld"
    ```

    PowerShell

    ```
    C:\greengrass\v2\bin\greengrass-cli deployment create --
    remove="com.example.HelloWorld"
    ```

> **Important**
>
> This step is required for you to deploy the component back to the core device after you upload it to AWS IoT Greengrass. Otherwise, the deployment fails with a version compatibility error because the local deployment specifies a different version of the component.

Run the following command and verify that the `com.example.HelloWorld` component doesn't appear in the list of components on your device.

Linux or Unix

```
sudo /greengrass/v2/bin/greengrass-cli component list
```

Windows Command Prompt (CMD)

```
C:\greengrass\v2\bin\greengrass-cli component list
```

PowerShell

```
C:\greengrass\v2\bin\greengrass-cli component list
```

Your Hello World component is complete, and you can now upload it to the AWS IoT Greengrass cloud service. Then, you can deploy the component to Greengrass core devices.

Step 5: Create your component in the AWS IoT Greengrass service

When you finish developing a component on your core device, you can upload it to the AWS IoT Greengrass service in the AWS Cloud. You can also directly create the component in the AWS IoT Greengrass console. AWS IoT Greengrass provides a component management service that hosts your components so that you can deploy them to individual devices or fleets of devices. To upload a component to the AWS IoT Greengrass service, you complete the following steps:

- Upload component artifacts to an S3 bucket.
- Add each artifact's Amazon Simple Storage Service (Amazon S3) URI to the component recipe.
- Create a component in AWS IoT Greengrass from the component recipe.

In this section, you complete these steps on your Greengrass core device to upload your Hello World component to the AWS IoT Greengrass service.

Create your component in AWS IoT Greengrass (console)

1. Use an S3 bucket in your AWS account to host AWS IoT Greengrass component artifacts. When you deploy the component to a core device, the device downloads the component's artifacts from the bucket.

   You can use an existing S3 bucket, or you can create a new bucket.

   a. In the Amazon S3 console, under **Buckets**, choose **Create bucket**.

   b. For **Bucket name**, enter a unique bucket name. For example, you can use **`greengrass-component-artifacts-`*`region`*`-`*`123456789012`***. Replace *`123456789012`* with your AWS account ID and *`region`* with the AWS Region that you use for this tutorial.

   c. For **AWS region**, select the AWS Region that you use for this tutorial.

   d. Choose **Create bucket**.

   e. Under **Buckets**, choose the bucket that you created, upload the `hello_world.py` script to the `artifacts/com.example.HelloWorld/1.0.0` folder in the bucket. For information about uploading objects to S3 buckets, see Uploading objects in the *Amazon Simple Storage Service User Guide*.

   f. Copy the S3 URI of the `hello_world.py` object in the S3 bucket. This URI should look similar to the following example. Replace amzn-s3-demo-bucket with the name of the S3 bucket.

   ```
   s3://amzn-s3-demo-bucket/artifacts/com.example.HelloWorld/1.0.0/hello_world.py
   ```

2. Allow the core device to access component artifacts in the S3 bucket.

   Each core device has a core device IAM role that allows it to interact with AWS IoT and send logs to the AWS Cloud. This device role doesn't allow access to S3 buckets by default, so you must create and attach a policy that allows the core device to retrieve component artifacts from the S3 bucket.

If your device's role already allows access to the S3 bucket, you can skip this step. Otherwise, create an IAM policy that allows access and attach it to the role, as follows:

a. In the IAM console navigation menu, choose **Policies**, and then choose **Create policy**.

b. On the **JSON** tab, replace the placeholder content with the following policy. Replace amzn-s3-demo-bucket with the name of the S3 bucket that contains component artifacts for the core device to download.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetObject"
      ],
      "Resource": "arn:aws:s3:::amzn-s3-demo-bucket/*"
    }
  ]
}
```

c. Choose **Next**.

d. In the **Policy details section**, for **Name**, enter **`MyGreengrassV2ComponentArtifactPolicy`**.

e. Choose **Create policy**.

f. In the IAM console navigation menu, choose **Role**, and then choose the name of the role for the core device. You specified this role name when you installed the AWS IoT Greengrass Core software. If you did not specify a name, the default is `GreengrassV2TokenExchangeRole`.

g. Under **Permissions**, choose **Add permissions**, then choose **Attach policies**.

h. On the **Add permissions** page, select the check box next to the `MyGreengrassV2ComponentArtifactPolicy` policy that you created, and then choose **Add permissions**.

3. Use the component recipe to create a component in the AWS IoT Greengrass console.

a. In the AWS IoT Greengrass console navigation menu, choose **Components**, and then choose **Create component**.

b. Under **Component information**, choose **Enter recipe as JSON**. The placeholder recipe should look similar to the following example.

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.HelloWorld",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "My first AWS IoT Greengrass component.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "Message": "world"
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "Run": "python3 -u {artifacts:path}/hello_world.py \"{configuration:/
Message}\""
      },
      "Artifacts": [
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.HelloWorld/1.0.0/hello_world.py"
        }
      ]
    },
    {
      "Platform": {
        "os": "windows"
      },
      "Lifecycle": {
        "Run": "py -3 -u {artifacts:path}/hello_world.py \"{configuration:/
Message}\""
      },
      "Artifacts": [
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.HelloWorld/1.0.0/hello_world.py"
        }
```

```
      ]
    }
  ]
}
```

c. Replace the placeholder URI in each `Artifacts` section with S3 URI of your `hello_world.py` object.

d. Choose **Create component**.

e. On the **com.example.HelloWorld** component page, verify that the **Status** of the component is **Deployable**.

Create your component in AWS IoT Greengrass (AWS CLI)

To upload your Hello World component

1. Use an S3 bucket in your AWS account to host AWS IoT Greengrass component artifacts. When you deploy the component to a core device, the device downloads the component's artifacts from the bucket.

   You can use an existing S3 bucket, or run the following command to create a bucket. This command creates a bucket with your AWS account ID and AWS Region to form a unique bucket name. Replace *123456789012* with your AWS account ID and *region* with the AWS Region that you use for this tutorial.

   ```
   aws s3 mb s3://greengrass-component-artifacts-123456789012-region
   ```

   The command outputs the following information if the request succeeds.

   ```
   make_bucket: greengrass-component-artifacts-123456789012-region
   ```

2. Allow the core device to access component artifacts in the S3 bucket.

   Each core device has a core device IAM role that allows it to interact with AWS IoT and send logs to the AWS Cloud. This device role doesn't allow access to S3 buckets by default, so you must create and attach a policy that allows the core device to retrieve component artifacts from the S3 bucket.

   If the core device's role already allows access to the S3 bucket, you can skip this step. Otherwise, create an IAM policy that allows access and attach it to the role, as follows:

a. Create a file called `component-artifact-policy.json` and copy the following JSON into the file. This policy allows access to all files in an S3 bucket. Replace amzn-s3-demo-bucket with the name of the S3 bucket.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetObject"
      ],
      "Resource": "arn:aws:s3:::amzn-s3-demo-bucket/*"
    }
  ]
}
```

b. Run the following command to create the policy from the policy document in `component-artifact-policy.json`.

Linux or Unix

```
aws iam create-policy \\
  --policy-name MyGreengrassV2ComponentArtifactPolicy \\
  --policy-document file://component-artifact-policy.json
```

Windows Command Prompt (CMD)

```
aws iam create-policy ^
  --policy-name MyGreengrassV2ComponentArtifactPolicy ^
  --policy-document file://component-artifact-policy.json
```

PowerShell

```
aws iam create-policy `
  --policy-name MyGreengrassV2ComponentArtifactPolicy `
  --policy-document file://component-artifact-policy.json
```

Copy the policy Amazon Resource Name (ARN) from the policy metadata in the output. You use this ARN to attach this policy to the core device role in the next step.

c. Run the following command to attach the policy to the core device role. Replace *GreengrassV2TokenExchangeRole* with the name of the role for the core device. You specified this role name when you installed the AWS IoT Greengrass Core software. Replace the policy ARN with the ARN from the previous step.

Linux or Unix

```
aws iam attach-role-policy \\
  --role-name GreengrassV2TokenExchangeRole \\
  --policy-arn
 arn:aws:iam::123456789012:policy/MyGreengrassV2ComponentArtifactPolicy
```

Windows Command Prompt (CMD)

```
aws iam attach-role-policy ^
  --role-name GreengrassV2TokenExchangeRole ^
  --policy-arn
 arn:aws:iam::123456789012:policy/MyGreengrassV2ComponentArtifactPolicy
```

PowerShell

```
aws iam attach-role-policy `
  --role-name GreengrassV2TokenExchangeRole `
  --policy-arn
 arn:aws:iam::123456789012:policy/MyGreengrassV2ComponentArtifactPolicy
```

If the command has no output, it succeeded. The core device can now access artifacts that you upload to this S3 bucket.

3. Upload the Hello World Python script artifact to the S3 bucket.

Run the following command to upload the script to the same path in the bucket where the script exists on your AWS IoT Greengrass core. Replace amzn-s3-demo-bucket with the name of the S3 bucket.

Linux or Unix

```
aws s3 cp \
  artifacts/com.example.HelloWorld/1.0.0/hello_world.py \
  s3://amzn-s3-demo-bucket/artifacts/com.example.HelloWorld/1.0.0/hello_world.py
```

Windows Command Prompt (CMD)

```
aws s3 cp ^
  artifacts/com.example.HelloWorld/1.0.0/hello_world.py ^
  s3://amzn-s3-demo-bucket/artifacts/com.example.HelloWorld/1.0.0/hello_world.py
```

PowerShell

```
aws s3 cp `
  artifacts/com.example.HelloWorld/1.0.0/hello_world.py `
  s3://amzn-s3-demo-bucket/artifacts/com.example.HelloWorld/1.0.0/hello_world.py
```

The command outputs a line that starts with `upload:` if the request succeeds.

4. Add the artifact's Amazon S3 URI to the component recipe.

   The Amazon S3 URI is composed of the bucket name and the path to the artifact object in the bucket. Your script artifact's Amazon S3 URI is the URI that you upload the artifact to in the previous step. This URI should look similar to the following example. Replace amzn-s3-demo-bucket with the name of the S3 bucket.

   ```
   s3://amzn-s3-demo-bucket/artifacts/com.example.HelloWorld/1.0.0/hello_world.py
   ```

   To add the artifact to the recipe, add a list of `Artifacts` that contains a structure with the Amazon S3 URI.

   JSON

   ```
   "Artifacts": [
     {
       "URI": "s3://amzn-s3-demo-bucket/artifacts/com.example.HelloWorld/1.0.0/
   hello_world.py"
     }
   ```

```
]
```

Open the recipe file in a text editor.

For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

```
nano recipes/com.example.HelloWorld-1.0.0.json
```

Add the artifact to the recipe. Your recipe file should look similar to the following example.

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.HelloWorld",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "My first AWS IoT Greengrass component.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "Message": "world"
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "Run": "python3 -u {artifacts:path}/hello_world.py \"{configuration:/
Message}\""
      },
      "Artifacts": [
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.HelloWorld/1.0.0/hello_world.py"
        }
      ]
    },
    {
      "Platform": {
        "os": "windows"
      },
```

```
      "Lifecycle": {
        "Run": "py -3 -u {artifacts:path}/hello_world.py \"{configuration:/
Message}\""
      },
      "Artifacts": [
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.HelloWorld/1.0.0/hello_world.py"
        }
      ]
    }
  ]
}
```

YAML

```
Artifacts:
  - URI: s3://amzn-s3-demo-bucket/artifacts/com.example.HelloWorld/1.0.0/
hello_world.py
```

Open the recipe file in a text editor.

For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

```
nano recipes/com.example.HelloWorld-1.0.0.yaml
```

Add the artifact to the recipe. Your recipe file should look similar to the following example.

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.HelloWorld
ComponentVersion: '1.0.0'
ComponentDescription: My first AWS IoT Greengrass component.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    Message: world
Manifests:
  - Platform:
      os: linux
```

```
    Lifecycle:
      Run: |
        python3 -u {artifacts:path}/hello_world.py "{configuration:/Message}"
    Artifacts:
      - URI: s3://amzn-s3-demo-bucket/artifacts/com.example.HelloWorld/1.0.0/
hello_world.py
  - Platform:
      os: windows
    Lifecycle:
      Run: |
        py -3 -u {artifacts:path}/hello_world.py "{configuration:/Message}"
    Artifacts:
      - URI: s3://amzn-s3-demo-bucket/artifacts/com.example.HelloWorld/1.0.0/
hello_world.py
```

5. Create a component resource in AWS IoT Greengrass from the recipe. Run the following command to create the component from the recipe, which you provide as a binary file.

   JSON

   ```
   aws greengrassv2 create-component-version --inline-recipe fileb://recipes/
   com.example.HelloWorld-1.0.0.json
   ```

   YAML

   ```
   aws greengrassv2 create-component-version --inline-recipe fileb://recipes/
   com.example.HelloWorld-1.0.0.yaml
   ```

   The response looks similar to the following example if the request succeeds.

   ```
   {
     "arn":
    "arn:aws:greengrass:region:123456789012:components:com.example.HelloWorld:versions:1.0.0",
     "componentName": "com.example.HelloWorld",
     "componentVersion": "1.0.0",
     "creationTimestamp": "Mon Nov 30 09:04:05 UTC 2020",
     "status": {
       "componentState": "REQUESTED",
       "message": "NONE",
       "errors": {}
     }
   ```

```
}
```

Copy the `arn` from the output to check the state of the component in the next step.

> **Note**
>
> You can also see your Hello World component in the AWS IoT Greengrass console on the **Components** page.

6. Verify that the component creates and is ready to be deployed. When you create a component, its state is `REQUESTED`. Then, AWS IoT Greengrass validates that the component is deployable. You can run the following command to query the component status and verify that your component is deployable. Replace the `arn` with the ARN from the previous step.

```
aws greengrassv2 describe-component --arn
 "arn:aws:greengrass:region:123456789012:components:com.example.HelloWorld:versions:1.0.0"
```

If the component validates, the response indicates that the component state is `DEPLOYABLE`.

```
{
  "arn":
 "arn:aws:greengrass:region:123456789012:components:com.example.HelloWorld:versions:1.0.0",
  "componentName": "com.example.HelloWorld",
  "componentVersion": "1.0.0",
  "creationTimestamp": "2020-11-30T18:04:05.823Z",
  "publisher": "Amazon",
  "description": "My first Greengrass component.",
  "status": {
    "componentState": "DEPLOYABLE",
    "message": "NONE",
    "errors": {}
  },
  "platforms": [
    {
      "os": "linux",
      "architecture": "all"
    }
  ]
}
```

Your Hello World component is now available in AWS IoT Greengrass. You can deploy it back to this Greengrass core device or to other core devices.

Step 6: Deploy your component

With AWS IoT Greengrass, you can deploy components to individual devices or groups of devices. When you deploy a component, AWS IoT Greengrass installs and runs that component's software on each target device. You specify which components to deploy and the configuration update to deploy for each component. You can also control how the deployment rolls out to the devices that the deployment targets. For more information, see Deploy AWS IoT Greengrass components to devices.

In this section, you deploy your Hello World component back to your Greengrass core device.

Deploy your component (console)

1. In the AWS IoT Greengrass console navigation menu, choose **Components**.
2. On the **Components** page, on the **My components** tab, choose **com.example.HelloWorld**.
3. On the **com.example.HelloWorld** page, choose **Deploy**.
4. From **Add to deployment**, choose **Create new deployment**, then choose **Next**.
5. On the **Specify target** page, do the following:
   a. In the **Name** box, enter **`Deployment for MyGreengrassCore`**.
   b. For **Deployment target**, choose **Core device**, and the name of the AWS IoT thing for your core device. The default value in this tutorial is *`MyGreengrassCore`*.
   c. Choose **Next**.
6. On the **Select components** page, under **My components**, verify that the **com.example.HelloWorld** component is selected, and choose **Next**.
7. On the **Configure components** page, choose **com.example.HelloWorld**, and do the following:
   a. Choose **Configure component**.
   b. Under **Configuration update**, in **Configuration to merge**, enter the following configuration.

```
{
  "Message": "universe"
```

```
}
```

This configuration update sets the Hello World `Message` parameter to `universe` for the device in this deployment.

c. Choose **Confirm**.

d. Choose **Next**.

8. On the **Configure advanced settings** page, keep the default configuration settings, and choose **Next**.
9. On the **Review** page, choose **Deploy**.
10. Verify that the deployment completes successfully. The deployment can take several minutes to complete. Check the Hello World log to verify the change. Run the following command on your Greengrass core device.

Linux or Unix

```
sudo tail -f /greengrass/v2/logs/com.example.HelloWorld.log
```

Windows Command Prompt (CMD)

```
type C:\greengrass\v2\\logs\\com.example.HelloWorld.log
```

PowerShell

```
gc C:\greengrass\v2\\logs\\com.example.HelloWorld.log -Tail 10 -Wait
```

You should see messages similar to the following example.

```
Hello, universe! Greetings from your first Greengrass component.
```

> **Note**
>
> If the log messages don't change, the deployment failed or didn't reach the core device. This can occur if your core device isn't connected to the internet or doesn't have permissions to retrieve artifacts from your S3 bucket. Run the following command on your core device to view the AWS IoT Greengrass Core software log file. This file includes logs from the Greengrass core device's deployment service.

Linux or Unix

```
sudo tail -f /greengrass/v2/logs/greengrass.log
```

Windows Command Prompt (CMD)

```
type C:\greengrass\v2\\logs\\greengrass.log
```

The `type` command writes the file's contents to the terminal. Run this command multiple times to observe changes in the file.

PowerShell

```
gc C:\greengrass\v2\\logs\\greengrass.log -Tail 10 -Wait
```

For more information, see Troubleshooting AWS IoT Greengrass V2.

Deploy your component (AWS CLI)

To deploy your Hello World component

1. On your development computer, create a file called `hello-world-deployment.json` and copy the following JSON into the file. This file defines the components and configurations to deploy.

```
{
  "components": {
    "com.example.HelloWorld": {
      "componentVersion": "1.0.0",
      "configurationUpdate": {
        "merge": "{\"Message\":\"universe\"}"
      }
    }
  }
}
```

This configuration file specifies to deploy version `1.0.0` of the Hello World component that you developed and published in the previous procedure. The `configurationUpdate` specifies to merge the component configuration in a JSON-encoded string. This configuration update sets the Hello World `Message` parameter to `universe` for the device in this deployment.

2. Run the following command to deploy the component to your Greengrass core device. You can deploy to things, which are individual devices, or thing groups, which are groups of devices. Replace *MyGreengrassCore* with the name of the AWS IoT thing for your core device.

   Linux or Unix

   ```
   aws greengrassv2 create-deployment \
     --target-arn "arn:aws:iot:region:account-id:thing/MyGreengrassCore" \
     --cli-input-json file://hello-world-deployment.json
   ```

   Windows Command Prompt (CMD)

   ```
   aws greengrassv2 create-deployment ^
     --target-arn "arn:aws:iot:region:account-id:thing/MyGreengrassCore" ^
     --cli-input-json file://hello-world-deployment.json
   ```

   PowerShell

   ```
   aws greengrassv2 create-deployment `
     --target-arn "arn:aws:iot:region:account-id:thing/MyGreengrassCore" `
     --cli-input-json file://hello-world-deployment.json
   ```

   The command outputs a response similar to the following example.

   ```
   {
     "deploymentId": "deb69c37-314a-4369-a6a1-3dff9fce73a9",
     "iotJobId": "b5d92151-6348-4941-8603-bdbfb3e02b75",
     "iotJobArn": "arn:aws:iot:region:account-id:job/b5d92151-6348-4941-8603-
   bdbfb3e02b75"
   }
   ```

3. Verify that the deployment completes successfully. The deployment can take several minutes to complete. Check the Hello World log to verify the change. Run the following command on your Greengrass core device.

   Linux or Unix

   ```
   sudo tail -f /greengrass/v2/logs/com.example.HelloWorld.log
   ```

   Windows Command Prompt (CMD)

   ```
   type C:\greengrass\v2\\logs\\com.example.HelloWorld.log
   ```

   PowerShell

   ```
   gc C:\greengrass\v2\\logs\\com.example.HelloWorld.log -Tail 10 -Wait
   ```

   You should see messages similar to the following example.

   ```
   Hello, universe! Greetings from your first Greengrass component.
   ```

   Note

   If the log messages don't change, the deployment failed or didn't reach the core device. This can occur if your core device isn't connected to the internet or doesn't have permissions to retrieve artifacts from your S3 bucket. Run the following command on your core device to view the AWS IoT Greengrass Core software log file. This file includes logs from the Greengrass core device's deployment service.

   Linux or Unix

   ```
   sudo tail -f /greengrass/v2/logs/greengrass.log
   ```

   Windows Command Prompt (CMD)

   ```
   type C:\greengrass\v2\\logs\\greengrass.log
   ```

The `type` command writes the file's contents to the terminal. Run this command multiple times to observe changes in the file.

PowerShell

```
gc C:\greengrass\v2\\logs\\greengrass.log -Tail 10 -Wait
```

For more information, see Troubleshooting AWS IoT Greengrass V2.

Next steps

You've completed this tutorial. The AWS IoT Greengrass Core software and your Hello World component run on your device. Also, your Hello World component is available in the AWS IoT Greengrass cloud service to deploy to other devices. For more information about the topics that this tutorial explores, see the following:

- Create AWS IoT Greengrass components
- Publish components to deploy to your core devices
- Deploy AWS IoT Greengrass components to devices

Setting up AWS IoT Greengrass core devices

Complete the tasks in this section to install, configure, and run the AWS IoT Greengrass Core software.

> **Note**
>
> This section describes advanced installation and configuration of the AWS IoT Greengrass Core software. These steps do not apply to nucleus lite.If you're a first-time user of AWS IoT Greengrass V2, we recommend that you first complete the getting started tutorial to set up a core device and explore the features of AWS IoT Greengrass.

Topics

- Supported platforms
- Device requirements
- Lambda function requirements
- Set up an AWS account
- Install the AWS IoT Greengrass Core software
- Run the AWS IoT Greengrass Core software
- Run AWS IoT Greengrass Core software in a Docker container
- Configure the AWS IoT Greengrass Core software
- Update the AWS IoT Greengrass Core software (OTA)
- Uninstall the AWS IoT Greengrass Core software

Supported platforms

- Greengrass nucleus supported platforms
- Greengrass nucleus lite supported platforms

Device requirements

- Greengrass nucleus device requirements

- Greengrass nucleus lite device requirements

Lambda function requirements

> **Important**
>
> Greengrass Lambda functions are currently not supported by Greengrass nucleus lite.

Your device must meet the following requirements to run Lambda functions:

- A Linux-based operating system.
- Your device must have the `mkfifo` shell command.
- Your device must run the programming language libraries that a Lambda function requires. You must install the required libraries on the device and add them to the PATH environment variable. Greengrass supports all Lambda supported versions of Python, Node.js, and Java runtimes. Greengrass doesn't apply any additional restrictions on deprecated Lambda runtime versions. For more information about AWS IoT Greengrass support for Lambda runtimes, see Run AWS Lambda functions.
- To run containerized Lambda functions, your device must meet the following requirements:
  - Linux kernel version 4.4 or later.
  - The kernel must support cgroups v1, and you must enable and mount the following cgroups:
    - The *memory* cgroup for AWS IoT Greengrass to set the memory limit for containerized Lambda functions.
    - The *devices* cgroup for containerized Lambda functions to access system devices or volumes.

    The AWS IoT Greengrass Core software doesn't support cgroups v2.

    To meet this requirement, boot the device with the following Linux kernel parameters.

    ```
    cgroup_enable=memory cgroup_memory=1 systemd.unified_cgroup_hierarchy=0
    ```

> **Tip**
>
> On a Raspberry Pi, edit the `/boot/cmdline.txt` file to set the device's kernel parameters.

- You must enable the following Linux kernel configurations on the device:
  - Namespace:
    - `CONFIG_IPC_NS`
    - `CONFIG_UTS_NS`
    - `CONFIG_USER_NS`
    - `CONFIG_PID_NS`
  - Cgroups:
    - `CONFIG_CGROUP_DEVICE`
    - `CONFIG_CGROUPS`
    - `CONFIG_MEMCG`
  - Others:
    - `CONFIG_POSIX_MQUEUE`
    - `CONFIG_OVERLAY_FS`
    - `CONFIG_HAVE_ARCH_SECCOMP_FILTER`
    - `CONFIG_SECCOMP_FILTER`
    - `CONFIG_KEYS`
    - `CONFIG_SECCOMP`
    - `CONFIG_SHMEM`

> **Tip**
>
> Check the documentation for your Linux distribution to learn how to verify and set Linux kernel parameters. You can also use AWS IoT Device Tester for AWS IoT Greengrass to verify that your device meets these requirements. For more information, see Using AWS IoT Device Tester for AWS IoT Greengrass V2.

Set up an AWS account

Install the AWS IoT Greengrass Core software

AWS IoT Greengrass extends AWS to edge devices so that they can act on the data they generate, while they use the AWS Cloud for management, analytics, and durable storage. Install the AWS IoT Greengrass Core software on edge devices to integrate with AWS IoT Greengrass and the AWS Cloud.

> **Important**
>
> Before you download and install the AWS IoT Greengrass Core software, check that your core device meets the requirements to install and run the AWS IoT Greengrass Core software v2.0.

The AWS IoT Greengrass Core software includes an installer that sets up your device as a Greengrass core device. When you run the installer, you can configure options, such as the root folder and the AWS Region to use. You can choose to have the installer create required AWS IoT and IAM resources for you. You can also choose to deploy local development tools to configure a device that you use for custom component development.

The AWS IoT Greengrass Core software requires the following AWS IoT and IAM resources to connect to the AWS Cloud and operate:

- An AWS IoT thing. When you register a device as an AWS IoT thing, that device can use a digital certificate to authenticate with AWS. This certificate allows the device to communicate with AWS IoT and AWS IoT Greengrass. For more information, see Device authentication and authorization for AWS IoT Greengrass.
- (Optional) An AWS IoT thing group. You use thing groups to manage fleets of Greengrass core devices. When you deploy software components to your devices, you can choose to deploy to individual devices or to groups of devices. You can add a device to a thing group to deploy that thing group's software components to the device. For more information, see Deploy AWS IoT Greengrass components to devices.
- An IAM role. Greengrass core devices use the AWS IoT Core credentials provider to authorize calls to AWS services with an IAM role. This role allows your device to interact with AWS IoT, send logs to Amazon CloudWatch Logs, and download custom component artifacts from Amazon Simple

Storage Service (Amazon S3). For more information, see Authorize core devices to interact with AWS services.

- An AWS IoT role alias. Greengrass core devices use the role alias to identify the IAM role to use. The role alias enables you to change the IAM role but keep the device configuration the same. For more information, see Authorizing direct calls to AWS services in the *AWS IoT Core Developer Guide*.

Choose one of the following options to install the AWS IoT Greengrass Core software on your core device.

- **Quick installation**

  Choose this option to set up a Greengrass core device in as few steps as possible. The installer creates the required AWS IoT and IAM resources for you. This option requires you to provide AWS credentials to the installer to create resources in your AWS account.

  You can't use this option to install behind a firewall or network proxy. If your devices are behind a firewall or network proxy, consider manual installation.

  For more information, see Install AWS IoT Greengrass Core software with automatic resource provisioning.

- **Manual installation**

  Choose this option to create the required AWS resources manually or to install behind a firewall or network proxy. By using a manual installation, you don't need to give the installer permission to create resources in your AWS account, because you create the required AWS IoT and IAM resources. You can also configure your device to connect on port 443 or through a network proxy. You can also configure the AWS IoT Greengrass Core software to use a private key and certificate that you store in a hardware security module (HSM), Trusted Platform Module (TPM), or another cryptographic element.

  For more information, see Install AWS IoT Greengrass Core software with manual resource provisioning.

- **Installation with AWS IoT fleet provisioning**

  Choose this option to create the required AWS resources from an AWS IoT fleet provisioning template. You might choose this option to create similar devices in a fleet, or if you manufacture devices that your customers later activate, such as vehicles or smart home devices. Devices

use claim certificates to authenticate and provision AWS resources, including an X.509 client certificate that the device uses to connect to the AWS Cloud for normal operation. You can embed or flash the claim certificates into the device's hardware during manufacturing, and you can use the same claim certificate and key to provision multiple devices. You can also configure devices to connect on port 443 or through a network proxy.

For more information, see Install AWS IoT Greengrass Core software with AWS IoT fleet provisioning.

- **Installation with custom provisioning**

  Choose this option to develop a custom Java application that provisions the required AWS resources. You might choose this option if you create your own X.509 client certificates or if you want more control over the provisioning process. AWS IoT Greengrass provides an interface that you can implement to exchange information between your custom provisioning application and the AWS IoT Greengrass Core software installer.

  For more information, see Install AWS IoT Greengrass Core software with custom resource provisioning.

AWS IoT Greengrass also provides containerized environments that run the AWS IoT Greengrass Core software. You can use a Dockerfile to run AWS IoT Greengrass in a Docker container.

Topics

- Install AWS IoT Greengrass Core software with automatic resource provisioning
- Install AWS IoT Greengrass Core software with manual resource provisioning
- Install AWS IoT Greengrass Core software with AWS IoT fleet provisioning
- Install AWS IoT Greengrass Core software with custom resource provisioning
- Installer arguments

Install AWS IoT Greengrass Core software with automatic resource provisioning

The AWS IoT Greengrass Core software includes an installer that sets up your device as a Greengrass core device. To set up a device quickly, the installer can provision the AWS IoT thing, AWS IoT thing group, IAM role, and AWS IoT role alias that the core device requires to operate. The installer can also deploy the local development tools to the core device, so you can use the

device to develop and test custom software components. The installer requires AWS credentials to provision these resources and create the deployment.

If you can't provide AWS credentials to the device, you can provision the AWS resources that the core device requires to operate. You can also deploy the development tools to a core device to use as a development device. This enables you to provide fewer permissions to the device when you run the installer. For more information, see Install AWS IoT Greengrass Core software with manual resource provisioning.

Important

Before you download the AWS IoT Greengrass Core software, check that your core device meets the requirements to install and run the AWS IoT Greengrass Core software v2.0.

Note

To install and run the AWS IoT Greengrass Core software as a non-root user, see Set up AWS IoT Greengrass V2 core devices as non-root.

Topics

- Set up the device environment
- Provide AWS credentials to the device
- Download the AWS IoT Greengrass Core software
- Install the AWS IoT Greengrass Core software

Set up the device environment

Follow the steps in this section to set up a Linux or Windows device to use as your AWS IoT Greengrass core device.

Set up a Linux device

To set up a Linux device for AWS IoT Greengrass V2

1. Install the Java runtime, which AWS IoT Greengrass Core software requires to run. We recommend that you use Amazon Corretto or OpenJDK long-term support versions. Version

8 or higher is required. The following commands show you how to install OpenJDK on your device.

- For Debian-based or Ubuntu-based distributions:

```
sudo apt install default-jdk
```

- For Red Hat-based distributions:

```
sudo yum install java-11-openjdk-devel
```

- For Amazon Linux 2:

```
sudo amazon-linux-extras install java-openjdk11
```

- For Amazon Linux 2023:

```
sudo dnf install java-11-amazon-corretto -y
```

When the installation completes, run the following command to verify that Java runs on your Linux device.

```
java -version
```

The command prints the version of Java that runs on the device. For example, on a Debian-based distribution, the output might look similar to the following sample.

```
openjdk version "11.0.9.1" 2020-11-04
OpenJDK Runtime Environment (build 11.0.9.1+1-post-Debian-1deb10u2)
OpenJDK 64-Bit Server VM (build 11.0.9.1+1-post-Debian-1deb10u2, mixed mode)
```

2. (Optional) Create the default system user and group that runs components on the device. You can also choose to let the AWS IoT Greengrass Core software installer create this user and group during installation with the `--component-default-user` installer argument. For more information, see Installer arguments.

```
sudo useradd --system --create-home ggc_user
sudo groupadd --system ggc_group
```

3. Verify that the user that runs the AWS IoT Greengrass Core software (typically `root`), has permission to run `sudo` with any user and any group.

   a. Run the following command to open the `/etc/sudoers` file.

   ```
   sudo visudo
   ```

   b. Verify that the permission for the user looks like the following example.

   ```
   root    ALL=(ALL:ALL) ALL
   ```

4. (Optional) To run containerized Lambda functions, you must enable cgroups v1, and you must enable and mount the *memory* and *devices* cgroups. If you don't plan to run containerized Lambda functions, you can skip this step.

   To enable these cgroups options, boot the device with the following Linux kernel parameters.

   ```
   cgroup_enable=memory cgroup_memory=1 systemd.unified_cgroup_hierarchy=0
   ```

   For information about viewing and setting kernel parameters for your device, see the documentation for your operating system and boot loader. Follow the instructions to permanently set the kernel parameters.

5. Install all other required dependencies on your device as indicated by the list of requirements in Device requirements.

Set up a Windows device

> **Note**
>
> This feature is available for v2.5.0 and later of the Greengrass nucleus component.

To set up a Windows device for AWS IoT Greengrass V2

1. Install the Java runtime, which AWS IoT Greengrass Core software requires to run. We recommend that you use Amazon Corretto or OpenJDK long-term support versions. Version 8 or higher is required.

2. Check whether Java is available on the PATH system variable, and add it if not. The LocalSystem account runs the AWS IoT Greengrass Core software, so you must add Java to the PATH system variable instead of the PATH user variable for your user. Do the following:

   a. Press the Windows key to open the start menu.

   b. Type **`environment variables`** to search for the system options from the start menu.

   c. In the start menu search results, choose **Edit the system environment variables** to open the **System properties** window.

   d. Choose **Environment variables...** to open the **Environment Variables** window.

   e. Under **System variables**, select **Path**, and then choose **Edit**. In the **Edit environment variable** window, you can view each path on a separate line.

   f. Check if the path to the Java installation's `bin` folder is present. The path might look similar to the following example.

   ```
   C:\\Program Files\\Amazon Corretto\\jdk11.0.13_8\\bin
   ```

   g. If the Java installation's `bin` folder is missing from **Path**, choose **New** to add it, and then choose **OK**.

3. Open the Windows Command Prompt (`cmd.exe`) as an administrator.

4. Create the default user in the LocalSystem account on the Windows device. Replace *`password`* with a secure password.

   ```
   net user /add ggc_user password
   ```

   Tip

   Depending on your Windows configuration, the user's password might be set to expire at a date in the future. To ensure your Greengrass applications continue to operate, track when the password expires, and update it before it expires. You can also set the user's password to never expire.

   - To check when a user and its password expire, run the following command.

     ```
     net user ggc_user | findstr /C:expires
     ```

   - To set a user's password to never expire, run the following command.

```
wmic UserAccount where "Name='ggc_user'" set PasswordExpires=False
```

- If you're using Windows 10 or later where the wmic command is deprecated, run the following PowerShell command.

```
Get-CimInstance -Query "SELECT * from Win32_UserAccount WHERE name =
 'ggc_user'" | Set-CimInstance -Property @{PasswordExpires="False"}
```

5. Download and install the PsExec utility from Microsoft on the device.
6. Use the PsExec utility to store the user name and password for the default user in the Credential Manager instance for the LocalSystem account. Replace *password* with the user's password that you set earlier.

```
psexec -s cmd /c cmdkey /generic:ggc_user /user:ggc_user /pass:password
```

If the **PsExec License Agreement** opens, choose **Accept** to agree to the license and run the command.

> **Note**
>
> On Windows devices, the LocalSystem account runs the Greengrass nucleus, and you must use the PsExec utility to store the default user information in the LocalSystem account. Using the Credential Manager application stores this information in the Windows account of the currently logged on user, instead of the LocalSystem account.

Provide AWS credentials to the device

Provide your AWS credentials to your device so that the installer can provision the required AWS resources. For more information about the required permissions, see Minimal IAM policy for installer to provision resources.

To provide AWS credentials to the device

- Provide your AWS credentials to the device so that the installer can provision the AWS IoT and IAM resources for your core device. To increase security, we recommend that you get

temporary credentials for an IAM role that allows only the minimum permissions necessary to provision. For more information, see Minimal IAM policy for installer to provision resources.

> **Note**
>
> The installer doesn't save or store your credentials.

On your device, do one of the following to retrieve credentials and make them available to the AWS IoT Greengrass Core software installer:

- (Recommended) Use temporary credentials from AWS IAM Identity Center

  a. Provide the access key ID, secret access key, and session token from the IAM Identity Center. For more information, see **Manual credential refresh** in Getting and refreshing temporary credentials in the *IAM Identity Center user guide*.

  b. Run the following commands to provide the credentials to the AWS IoT Greengrass Core software.

  Linux or Unix

  ```
  export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
  export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
  export AWS_SESSION_TOKEN=AQoDYXdzEJr1K...o5OytwEXAMPLE=
  ```

  Windows Command Prompt (CMD)

  ```
  set AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
  set AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
  set AWS_SESSION_TOKEN=AQoDYXdzEJr1K...o5OytwEXAMPLE=
  ```

  PowerShell

  ```
  $env:AWS_ACCESS_KEY_ID="AKIAIOSFODNN7EXAMPLE"
  $env:AWS_SECRET_ACCESS_KEY="wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"
  $env:AWS_SESSION_TOKEN="AQoDYXdzEJr1K...o5OytwEXAMPLE="
  ```

- Use temporary security credentials from an IAM role:

a. Provide the access key ID, secret access key, and session token from an IAM role that you assume. For more information about how to retrieve these credentials, see Requesting temporary security credentials in the *IAM User Guide*.

b. Run the following commands to provide the credentials to the AWS IoT Greengrass Core software.

Linux or Unix

```
export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
export AWS_SESSION_TOKEN=AQoDYXdzEJr1K...o5OytwEXAMPLE=
```

Windows Command Prompt (CMD)

```
set AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
set AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
set AWS_SESSION_TOKEN=AQoDYXdzEJr1K...o5OytwEXAMPLE=
```

PowerShell

```
$env:AWS_ACCESS_KEY_ID="AKIAIOSFODNN7EXAMPLE"
$env:AWS_SECRET_ACCESS_KEY="wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"
$env:AWS_SESSION_TOKEN="AQoDYXdzEJr1K...o5OytwEXAMPLE="
```

- Use long-term credentials from an IAM user:

a. Provide the access key ID and secret access key for your IAM user. You can create an IAM user for provisioning that you later delete. For the IAM policy to give the user, see Minimal IAM policy for installer to provision resources. For more information about how to retrieve long-term credentials, see Managing access keys for IAM users in the *IAM User Guide*.

b. Run the following commands to provide the credentials to the AWS IoT Greengrass Core software.

Linux or Unix

```
export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

Windows Command Prompt (CMD)

```
set AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
set AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

PowerShell

```
$env:AWS_ACCESS_KEY_ID="AKIAIOSFODNN7EXAMPLE"
$env:AWS_SECRET_ACCESS_KEY="wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"
```

c. (Optional) If you created an IAM user to provision your Greengrass device, delete the user.

d. (Optional) If you used the access key ID and secret access key from an existing IAM user, update the keys for the user so that they are no longer valid. For more information, see Updating access keys in the *AWS Identity and Access Management user guide*.

Download the AWS IoT Greengrass Core software

You can download the latest version of the AWS IoT Greengrass Core software from the following location:

- https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip

> **Note**
>
> You can download a specific version of the AWS IoT Greengrass Core software from the following location. Replace *version* with the version to download.
>
> ```
> https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-version.zip
> ```

To download the AWS IoT Greengrass Core software

1. On your core device, download the AWS IoT Greengrass Core software to a file named `greengrass-nucleus-latest.zip`.

Linux or Unix

```
curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-
latest.zip > greengrass-nucleus-latest.zip
```

Windows Command Prompt (CMD)

```
curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-
latest.zip > greengrass-nucleus-latest.zip
```

PowerShell

```
iwr -Uri https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-
latest.zip -OutFile greengrass-nucleus-latest.zip
```

By downloading this software, you agree to the Greengrass Core Software License Agreement.

2. (Optional) To verify the Greengrass nucleus software signature

> **Note**
>
> This feature is available with Greengrass nucleus version 2.9.5 and later.

a. Use the following command to verify your Greengrass nucleus artifact's signature:

Linux or Unix

```
jarsigner -verify -certs -verbose greengrass-nucleus-latest.zip
```

Windows Command Prompt (CMD)

The file name might look different depending on the JDK version you install. Replace *jdk17.0.6_10* with the JDK version you installed.

```
"C:\\Program Files\\Amazon Corretto\\jdk17.0.6_10\\bin\\jarsigner.exe" -
verify -certs -verbose greengrass-nucleus-latest.zip
```

PowerShell

The file name might look different depending on the JDK version you install. Replace *jdk17.0.6_10* with the JDK version you installed.

```
'C:\\Program Files\\Amazon Corretto\\jdk17.0.6_10\\bin\\jarsigner.exe' -
verify -certs -verbose greengrass-nucleus-latest.zip
```

b. The `jarsigner` invocation yields output that indicates the results of the verification.

   i. If the Greengrass nucleus zip file is signed, the output contains the following statement:

   ```
   jar verified.
   ```

   ii. If the Greengrass nucleus zip file isn't signed, the output contains the following statement:

   ```
   jar is unsigned.
   ```

c. If you provided the Jarsigner `-certs` option along with `-verify` and `-verbose` options, the output also includes detailed signer certificate information.

3. Unzip the AWS IoT Greengrass Core software to a folder on your device. Replace *GreengrassInstaller* with the folder that you want to use.

   Linux or Unix

   ```
   unzip greengrass-nucleus-latest.zip -d GreengrassInstaller && rm greengrass-
   nucleus-latest.zip
   ```

   Windows Command Prompt (CMD)

   ```
   mkdir GreengrassInstaller && tar -xf greengrass-nucleus-latest.zip -
   C GreengrassInstaller && del greengrass-nucleus-latest.zip
   ```

   PowerShell

   ```
   Expand-Archive -Path greengrass-nucleus-latest.zip -DestinationPath .\
   \GreengrassInstaller
   ```

```
rm greengrass-nucleus-latest.zip
```

4. (Optional) Run the following command to see the version of the AWS IoT Greengrass Core software.

```
java -jar ./GreengrassInstaller/lib/Greengrass.jar --version
```

> **Important**
>
> If you install a version of the Greengrass nucleus earlier than v2.4.0, don't remove this folder after you install the AWS IoT Greengrass Core software. The AWS IoT Greengrass Core software uses the files in this folder to run.
> If you downloaded the latest version of the software, you install v2.4.0 or later, and you can remove this folder after you install the AWS IoT Greengrass Core software.

Install the AWS IoT Greengrass Core software

Run the installer with arguments that specify to do the following:

- Create the AWS resources that the core device requires to operate.
- Specify to use the `ggc_user` system user to run software components on the core device. On Linux devices, this command also specifies to use the `ggc_group` system group, and the installer creates the system user and group for you.
- Set up the AWS IoT Greengrass Core software as a system service that runs at boot. On Linux devices, this requires the Systemd init system.

> **Important**
>
> On Windows core devices, you must set up the AWS IoT Greengrass Core software as a system service.

To set up a development device with local development tools, specify the `--deploy-dev-tools true` argument. The local development tools can take up to a minute to deploy after the installation completes.

For more information about the arguments that you can specify, see Installer arguments.

> **Note**
>
> If you are running AWS IoT Greengrass on a device with limited memory, you can control the amount of memory that AWS IoT Greengrass Core software uses. To control memory allocation, you can set JVM heap size options in the `jvmOptions` configuration parameter in your nucleus component. For more information, see Control memory allocation with JVM options.

To install the AWS IoT Greengrass Core software

1. Run the AWS IoT Greengrass Core installer. Replace argument values in your command as follows.

   > **Note**
   >
   > Windows has a path length limitation of 260 characters. If you are using Windows, use a root folder like `C:\greengrass\v2` or `D:\greengrass\v2` to keep the Greengrass components paths under the 260 character limit.

   a. */greengrass/v2* or *C:\greengrass\v2*: The path to the root folder to use to install the AWS IoT Greengrass Core software.

   b. *GreengrassInstaller*. The path to the folder where you unpacked the AWS IoT Greengrass Core software installer.

   c. *region*. The AWS Region in which to find or create resources.

   d. *MyGreengrassCore*. The name of the AWS IoT thing for your Greengrass core device. If the thing doesn't exist, the installer creates it. The installer downloads the certificates to authenticate as the AWS IoT thing. For more information, see Device authentication and authorization for AWS IoT Greengrass.

      > **Note**
      >
      > The thing name can't contain colon (`:`) characters.

e. *MyGreengrassCoreGroup*. The name of AWS IoT thing group for your Greengrass core device. If the thing group doesn't exist, the installer creates it and adds the thing to it. If the thing group exists and has an active deployment, the core device downloads and runs the software that the deployment specifies.

> **Note**
>
> The thing group name can't contain colon (`:`) characters.

f. *GreengrassV2IoTThingPolicy*. The name of the AWS IoT policy that allows the Greengrass core devices to communicate with AWS IoT and AWS IoT Greengrass. If the AWS IoT policy doesn't exist, the installer creates a permissive AWS IoT policy with this name. You can restrict this policy's permissions for you use case. For more information, see Minimal AWS IoT policy for AWS IoT Greengrass V2 core devices.

g. *GreengrassV2TokenExchangeRole*. The name of the IAM role that allows the Greengrass core device to get temporary AWS credentials. If the role doesn't exist, the installer creates it and creates and attaches a policy named *GreengrassV2TokenExchangeRole*`Access`. For more information, see Authorize core devices to interact with AWS services.

h. *GreengrassCoreTokenExchangeRoleAlias*. The alias to the IAM role that allows the Greengrass core device to get temporary credentials later. If the role alias doesn't exist, the installer creates it and points it to the IAM role that you specify. For more information, see Authorize core devices to interact with AWS services.

Linux or Unix

```
sudo -E java -Droot="/greengrass/v2" -Dlog.store=FILE \
  -jar ./GreengrassInstaller/lib/Greengrass.jar \
  --aws-region region \
  --thing-name MyGreengrassCore \
  --thing-group-name MyGreengrassCoreGroup \
  --thing-policy-name GreengrassV2IoTThingPolicy \
  --tes-role-name GreengrassV2TokenExchangeRole \
  --tes-role-alias-name GreengrassCoreTokenExchangeRoleAlias \
  --component-default-user ggc_user:ggc_group \
  --provision true \
  --setup-system-service true
```

Windows Command Prompt (CMD)

```
java -Droot="C:\greengrass\v2" "-Dlog.store=FILE" ^
  -jar ./GreengrassInstaller/lib/Greengrass.jar ^
  --aws-region region ^
  --thing-name MyGreengrassCore ^
  --thing-group-name MyGreengrassCoreGroup ^
  --thing-policy-name GreengrassV2IoTThingPolicy ^
  --tes-role-name GreengrassV2TokenExchangeRole ^
  --tes-role-alias-name GreengrassCoreTokenExchangeRoleAlias ^
  --component-default-user ggc_user ^
  --provision true ^
  --setup-system-service true
```

PowerShell

```
java -Droot="C:\greengrass\v2" "-Dlog.store=FILE" `
  -jar ./GreengrassInstaller/lib/Greengrass.jar `
  --aws-region region `
  --thing-name MyGreengrassCore `
  --thing-group-name MyGreengrassCoreGroup `
  --thing-policy-name GreengrassV2IoTThingPolicy `
  --tes-role-name GreengrassV2TokenExchangeRole `
  --tes-role-alias-name GreengrassCoreTokenExchangeRoleAlias `
  --component-default-user ggc_user `
  --provision true `
  --setup-system-service true
```

Important

On Windows core devices, you must specify `--setup-system-service true` to set up the AWS IoT Greengrass Core software as a system service.

The installer prints the following messages if it succeeds:

- If you specify `--provision`, the installer prints `Successfully configured Nucleus with provisioned resource details` if it configured the resources successfully.

- If you specify `--deploy-dev-tools`, the installer prints `Configured Nucleus to deploy aws.greengrass.Cli component` if it created the deployment successfully.
- If you specify `--setup-system-service true`, the installer prints `Successfully set up Nucleus as a system service` if it set up and ran the software as a service.
- If you don't specify `--setup-system-service true`, the installer prints `Launched Nucleus successfully` if it succeeded and ran the software.

2. Skip this step if you installed Greengrass nucleus v2.0.4 or later. If you downloaded the latest version of the software, you installed v2.0.4 or later.

   Run the following command to set the required file permissions for your AWS IoT Greengrass Core software root folder. Replace */greengrass/v2* with the root folder that you specified in your installation command, and replace */greengrass* with the parent folder for your root folder.

   ```
   sudo chmod 755 /greengrass/v2 && sudo chmod 755 /greengrass
   ```

If you installed the AWS IoT Greengrass Core software as a system service, the installer runs the software for you. Otherwise, you must run the software manually. For more information, see Run the AWS IoT Greengrass Core software.

> **Note**
>
> By default, the IAM role that the installer creates doesn't allow access to component artifacts in S3 buckets. To deploy custom components that define artifacts in Amazon S3, you must add permissions to the role to allow your core device to retrieve component artifacts. For more information, see Allow access to S3 buckets for component artifacts. If you don't yet have an S3 bucket for component artifacts, you can add these permissions later after you create a bucket.

> **Note**
>
> When the AWS IoT Greengrass Core software connects to the cloud, your device will be recognized as a Core device.

For more information about how to configure and use the software and AWS IoT Greengrass, see the following:

- Configure the AWS IoT Greengrass Core software
- Develop AWS IoT Greengrass components
- Deploy AWS IoT Greengrass components to devices
- Greengrass Command Line Interface

Install AWS IoT Greengrass Core software with manual resource provisioning

The AWS IoT Greengrass Core software includes an installer that sets up your device as a Greengrass core device. To set up a device manually, you can create the required AWS IoT and IAM resources for the device to use. If you create these resources manually, you don't need to provide AWS credentials to the installer.

When you manually install the AWS IoT Greengrass Core software, you can also configure the device to use a network proxy or connect to AWS on port 443. You might need to specify these configuration options if your device runs behind a firewall or a network proxy, for example. For more information, see Connect on port 443 or through a network proxy.

You can also configure the AWS IoT Greengrass Core software to use a hardware security module (HSM) through the PKCS#11 interface. This feature enables you to securely store private key and certificate files so that they aren't exposed or duplicated in software. You can store private keys and certificates on a hardware module such as an HSM, a Trusted Platform Module (TPM), or another cryptographic element. This feature is available on Linux devices only. For more information about hardware security and requirements to use it, see Hardware security integration.

> **Important**
>
> Before you download the AWS IoT Greengrass Core software, check that your core device meets the requirements to install and run the AWS IoT Greengrass Core software v2.0.

Topics

- Retrieve AWS IoT endpoints

- Create an AWS IoT thing
- Create the thing certificate
- Configure the thing certificate
- Create a token exchange role
- Download certificates to the device
- Set up the device environment
- Download the AWS IoT Greengrass Core software
- Install the AWS IoT Greengrass Core software

Retrieve AWS IoT endpoints

Get the AWS IoT endpoints for your AWS account, and save them to use later. Your device uses these endpoints to connect to AWS IoT. Do the following:

1. Get the AWS IoT data endpoint for your AWS account.

   ```
   aws iot describe-endpoint --endpoint-type iot:Data-ATS
   ```

   The response looks similar to the following example, if the request succeeds.

   ```
   {
     "endpointAddress": "device-data-prefix-ats.iot.us-west-2.amazonaws.com"
   }
   ```

2. Get the AWS IoT credentials endpoint for your AWS account.

   ```
   aws iot describe-endpoint --endpoint-type iot:CredentialProvider
   ```

   The response looks similar to the following example, if the request succeeds.

   ```
   {
     "endpointAddress": "device-credentials-prefix.credentials.iot.us-
   west-2.amazonaws.com"
   }
   ```

Create an AWS IoT thing

AWS IoT *things* represent devices and logical entities that connect to AWS IoT. Greengrass core devices are AWS IoT things. When you register a device as an AWS IoT thing, that device can use a digital certificate to authenticate with AWS.

In this section, you create an AWS IoT thing that represents your device.

To create an AWS IoT thing

1. Create an AWS IoT thing for your device. On your development computer, run the following command.

   - Replace *MyGreengrassCore* with the thing name to use. This name is also the name of your Greengrass core device.

     > **Note**
     >
     > The thing name can't contain colon (:) characters.

   ```
   aws iot create-thing --thing-name MyGreengrassCore
   ```

   The response looks similar to the following example, if the request succeeds.

   ```
   {
     "thingName": "MyGreengrassCore",
     "thingArn": "arn:aws:iot:us-west-2:123456789012:thing/MyGreengrassCore",
     "thingId": "8cb4b6cd-268e-495d-b5b9-1713d71dbf42"
   }
   ```

2. (Optional) Add the AWS IoT thing to a new or existing thing group. You use thing groups to manage fleets of Greengrass core devices. When you deploy software components to your devices, you can target individual devices or groups of devices. You can add a device to a thing group with an active Greengrass deployment to deploy that thing group's software components to the device. Do the following:

   a. (Optional) Create an AWS IoT thing group.

      - Replace *MyGreengrassCoreGroup* with the name of the thing group to create.

> **Note**
>
> The thing group name can't contain colon (:) characters.

```
aws iot create-thing-group --thing-group-name MyGreengrassCoreGroup
```

The response looks similar to the following example, if the request succeeds.

```
{
  "thingGroupName": "MyGreengrassCoreGroup",
  "thingGroupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/
MyGreengrassCoreGroup",
  "thingGroupId": "4df721e1-ff9f-4f97-92dd-02db4e3f03aa"
}
```

b. Add the AWS IoT thing to a thing group.

- Replace *MyGreengrassCore* with the name of your AWS IoT thing.
- Replace *MyGreengrassCoreGroup* with the name of the thing group.

```
aws iot add-thing-to-thing-group --thing-name MyGreengrassCore --thing-group-
name MyGreengrassCoreGroup
```

The command doesn't have any output if the request succeeds.

Create the thing certificate

When you register a device as an AWS IoT thing, that device can use a digital certificate to authenticate with AWS. This certificate allows the device to communicate with AWS IoT and AWS IoT Greengrass.

In this section, you create and download certificates that your device can use to connect to AWS.

If you want to configure the AWS IoT Greengrass Core software to use a hardware security module (HSM) to securely store the private key and certificate, follow the steps to create the certificate from a private key in an HSM. Otherwise, follow the steps to create the certificate and private key

in the AWS IoT service. The hardware security feature is available on Linux devices only. For more information about hardware security and requirements to use it, see Hardware security integration.

Create the certificate and private key in the AWS IoT service

To create the thing certificate

1. Create a folder where you download the certificates for the AWS IoT thing.

```
mkdir greengrass-v2-certs
```

2. Create and download the certificates for the AWS IoT thing.

```
aws iot create-keys-and-certificate --set-as-active --certificate-pem-outfile
 greengrass-v2-certs/device.pem.crt --public-key-outfile greengrass-v2-certs/
public.pem.key --private-key-outfile greengrass-v2-certs/private.pem.key
```

   The response looks similar to the following example, if the request succeeds.

```
{
  "certificateArn": "arn:aws:iot:us-west-2:123456789012:cert/
aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4",
  "certificateId":
 "aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4",
  "certificatePem": "-----BEGIN CERTIFICATE-----
MIICiTCCAfICCQD6m7oRw0uXOjANBgkqhkiG9w
 0BAQUFADCBiDELMAkGA1UEBhMCVVMxCzAJBgNVBAgTAldBMRAwDgYDVQQHEwdTZ
 WF0dGxlMQ8wDQYDVQQKEwZBbWF6b24xFDASBgNVBAsTC0lBTSBDb25zb2xlMRIw
 EAYDVQQDEwlUZXN0Q2lsYWMxHzAdBgkqhkiG9w0BCQEWEG5vb25lQGFtYXpvbi5
 jb20wHhcNMTEwNDI1MjA0NTIxWhcNMTIwNDI0MjA0NTIxWjCBiDELMAkGA1UEBh
 MCVVMxCzAJBgNVBAgTAldBMRAwDgYDVQQHEwdTZWF0dGxlMQ8wDQYDVQQKEwZBb
 WF6b24xFDASBgNVBAsTC0lBTSBDb25zb2xlMRIwEAYDVQQDEwlUZXN0Q2lsYWMx
 HzAdBgkqhkiG9w0BCQEWEG5vb25lQGFtYXpvbi5jb20wgZ8wDQYJKoZIhvcNAQE
 BBQADgY0AMIGJAoGBAMaK0dn+a4GmWIWJ21uUSfwfEvySWtC2XADZ4nB+BLYgVI
 k60CpiwsZ3G93vUEIO3IyNoH/f0wYK8m9TrDHudUZg3qX4waLG5M43q7Wgc/MbQ
 ITxOUSQv7c7ugFFDzQGBzZswY6786m86gpEIbb3OhjZnzcvQAaRHhdlQWIMm2nr
 AgMBAAEwDQYJKoZIhvcNAQEFBQADgYEAtCu4nUhVVxYUntneD9+h8Mg9q6q+auN
 KyExzyLwaxlAoo7TJHidbtS4J5iNmZgXL0FkbFFBjvSfpJIlJ00zbhNYS5f6Guo
 EDmFJl0ZxBHjJnyp378OD8uTs7fLvjx79LjSTbNYiytVbZPQUQ5Yaxu2jXnimvw
 3rrszlaEXAMPLE=
-----END CERTIFICATE-----",
  "keyPair": {
```

```
    "PublicKey": "-----BEGIN PUBLIC KEY-----\
MIIBIjANBgkqhkEXAMPLEQEFAAOCAQ8AMIIBCgKCAQEAEXAMPLE1nnyJwKSMHw4h\
MMEXAMPLEuuN/dMAS3fyce8DW/4+EXAMPLEyjmoF/YVF/gHr99VEEXAMPLE5VF13\
59VK7cEXAMPLE67GK+y+jikqXOgHh/xJTwo
+sGpWEXAMPLEDz18xOd2ka4tCzuWEXAMPLEahJbYkCPUBSU8opVkR7qkEXAMPLE1DR6sx2HocliOOLtu6Fkw91swQWE
\\GB3ZPrNh0PzQYvjUStZeccyNCx2EXAMPLEvp9mQOUXP6plfgxwKRX2fEXAMPLEDa\
hJLXkX3rHU2xbxJSq7D+XEXAMPLEcw+LyFhI5mgFRl88eGdsAEXAMPLElnI9EesG\
FQIDAQAB\
-----END PUBLIC KEY-----\
",
    "PrivateKey": "-----BEGIN RSA PRIVATE KEY-----\
key omitted for security reasons\
-----END RSA PRIVATE KEY-----\
"
  }
}
```

Save the certificate's Amazon Resource Name (ARN) to use to configure the certificate later.

Create the certificate from a private key in an HSM

> **Note**
>
> This feature is available for v2.5.3 and later of the Greengrass nucleus component. AWS IoT Greengrass doesn't currently support this feature on Windows core devices.

To create the thing certificate

1. On the core device, initialize a PKCS#11 token in the HSM, and generate a private key. The private key must be an RSA key with an RSA-2048 key size (or larger) or an ECC key.

   > **Note**
   >
   > To use a hardware security module with ECC keys, you must use Greengrass nucleus v2.5.6 or later.
   > To use a hardware security module and secret manager, you must use a hardware security module with RSA keys.

Check the documentation for your HSM to learn how to initialize the token and generate the private key. If your HSM supports object IDs, specify an object ID when you generate the private key. Save the slot ID, user PIN, object label, object ID (if your HSM uses one) that you specify when you initialize the token and generate the private key. You use these values later when you import the thing certificate to the HSM and configure the AWS IoT Greengrass Core software.

2. Create a certificate signing request (CSR) from the private key. AWS IoT uses this CSR to create a thing certificate for the private key that you generated in the HSM. For information about how to create a CSR from the private key, see the documentation for your HSM. The CSR is a file, such as `iotdevicekey.csr`.
3. Copy the CSR from the device to your development computer. If SSH and SCP are enabled on the development computer and the device, you can use the `scp` command on your development computer to transfer the CSR. Replace *`device-ip-address`* with the IP address of your device, and replace *`~/iotdevicekey.csr`* with the path to the CSR file on the device.

   ```
   scp device-ip-address:~/iotdevicekey.csr iotdevicekey.csr
   ```

4. On your development computer, create a folder where you download the certificate for the AWS IoT thing.

   ```
   mkdir greengrass-v2-certs
   ```

5. Use the CSR file to create and download the certificate for the AWS IoT thing to your development computer.

   ```
   aws iot create-certificate-from-csr --set-as-active --certificate-signing-
   request=file://iotdevicekey.csr --certificate-pem-outfile greengrass-v2-certs/
   device.pem.crt
   ```

   The response looks similar to the following example, if the request succeeds.

   ```
   {
     "certificateArn": "arn:aws:iot:us-west-2:123456789012:cert/
   aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4",
     "certificateId":
    "aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4",
     "certificatePem": "-----BEGIN CERTIFICATE-----
   ```

```
MIICiTCCAfICCQD6m7oRw0uXOjANBgkqhkiG9w
 0BAQUFADCBiDELMAkGA1UEBhMCVVMxCzAJBgNVBAgTAldBMRAwDgYDVQQHEwdTZ
 WF0dGxlMQ8wDQYDVQQKEwZBbWF6b24xFDASBgNVBAsTC0lBTSBDb25zb2xlMRIw
 EAYDVQQDEwlUZXN0Q2lsYWMxHzAdBgkqhkiG9w0BCQEWEG5vb25lQGFtYXpvbi5
 jb20wHhcNMTEwNDI1MjA0NTIxWhcNMTIwNDI0MjA0NTIxWjCBiDELMAkGA1UEBh
 MCVVMxCzAJBgNVBAgTAldBMRAwDgYDVQQHEwdTZWF0dGxlMQ8wDQYDVQQKEwZBb
 WF6b24xFDASBgNVBAsTC0lBTSBDb25zb2xlMRIwEAYDVQQDEwlUZXN0Q2lsYWMx
 HzAdBgkqhkiG9w0BCQEWEG5vb25lQGFtYXpvbi5jb20wgZ8wDQYJKoZIhvcNAQE
 BBQADgY0AMIGJAoGBAMaK0dn+a4GmWIWJ21uUSfwfEvySWtC2XADZ4nB+BLYgVI
 k60CpiwsZ3G93vUEIO3IyNoH/f0wYK8m9TrDHudUZg3qX4waLG5M43q7Wgc/MbQ
 ITxOUSQv7c7ugFFDzQGBzZswY6786m86gpEIbb3OhjZnzcvQAaRHhdlQWIMm2nr
 AgMBAAEwDQYJKoZIhvcNAQEFBQADgYEAtCu4nUhVVxYUntneD9+h8Mg9q6q+auN
 KyExzyLwaxlAoo7TJHidbtS4J5iNmZgXL0FkbFFBjvSfpJIlJ00zbhNYS5f6Guo
 EDmFJl0ZxBHjJnyp378OD8uTs7fLvjx79LjSTbNYiytVbZPQUQ5Yaxu2jXnimvw
 3rrszlaEXAMPLE=
-----END CERTIFICATE-----"
}
```

Save the certificate's ARN to use to configure the certificate later.

Configure the thing certificate

Attach the thing certificate to the AWS IoT thing that you created earlier, and add an AWS IoT policy to the certificate to define the AWS IoT permissions for the core device.

To configure the thing's certificate

1. Attach the certificate to the AWS IoT thing.

   - Replace *MyGreengrassCore* with the name of your AWS IoT thing.
   - Replace the certificate Amazon Resource Name (ARN) with the ARN of the certificate that you created in the previous step.

```
aws iot attach-thing-principal --thing-name MyGreengrassCore
 --principal arn:aws:iot:us-west-2:123456789012:cert/
aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4
```

   The command doesn't have any output if the request succeeds.

2. Create and attach an AWS IoT policy that defines the AWS IoT permissions for your Greengrass core device. The following policy allows access to all MQTT topics and Greengrass operations, so your device works with custom applications and future changes that require new Greengrass operations. You can restrict this policy down based on your use case. For more information, see Minimal AWS IoT policy for AWS IoT Greengrass V2 core devices.

   If you have set up a Greengrass core device before, you can attach its AWS IoT policy instead of creating a new one.

   Do the following:

   a. Create a file that contains the AWS IoT policy document that Greengrass core devices require.

      For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

      ```
      nano greengrass-v2-iot-policy.json
      ```

      Copy the following JSON into the file.

      ```
      {
        "Version": "2012-10-17",
        "Statement": [
          {
            "Effect": "Allow",
            "Action": [
              "iot:Publish",
              "iot:Subscribe",
              "iot:Receive",
              "iot:Connect",
              "greengrass:*"
            ],
            "Resource": [
              "*"
            ]
          }
        ]
      }
      ```

   b. Create an AWS IoT policy from the policy document.

- Replace *GreengrassV2IoTThingPolicy* with the name of the policy to create.

```
aws iot create-policy --policy-name GreengrassV2IoTThingPolicy --policy-
document file://greengrass-v2-iot-policy.json
```

The response looks similar to the following example, if the request succeeds.

```
{
  "policyName": "GreengrassV2IoTThingPolicy",
  "policyArn": "arn:aws:iot:us-west-2:123456789012:policy/
GreengrassV2IoTThingPolicy",
  "policyDocument": "{
    \\"Version\\": \\"2012-10-17       \\",
    \\"Statement\\": [
      {
        \\"Effect\\": \\"Allow\\",
        \\"Action\\": [
          \\"iot:Publish\\",
          \\"iot:Subscribe\\",
          \\"iot:Receive\\",
          \\"iot:Connect\\",
          \\"greengrass:*\\"
        ],
        \\"Resource\\": [
          \\"*\\"
        ]
      }
    ]
  }",
  "policyVersionId": "1"
}
```

c. Attach the AWS IoT policy to the AWS IoT thing's certificate.

- Replace *GreengrassV2IoTThingPolicy* with the name of the policy to attach.
- Replace the target ARN with the ARN of the certificate for your AWS IoT thing.

```
aws iot attach-policy --policy-name GreengrassV2IoTThingPolicy
 --target arn:aws:iot:us-west-2:123456789012:cert/
aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4
```

The command doesn't have any output if the request succeeds.

Create a token exchange role

Greengrass core devices use an IAM service role, called the *token exchange role*, to authorize calls to AWS services. The device uses the AWS IoT credentials provider to get temporary AWS credentials for this role, which allows the device to interact with AWS IoT, send logs to Amazon CloudWatch Logs, and download custom component artifacts from Amazon S3. For more information, see Authorize core devices to interact with AWS services.

You use an AWS IoT *role alias* to configure the token exchange role for Greengrass core devices. Role aliases enable you to change the token exchange role for a device but keep the device configuration the same. For more information, see Authorizing direct calls to AWS services in the *AWS IoT Core Developer Guide*.

In this section, you create a token exchange IAM role and an AWS IoT role alias that points to the role. If you have already set up a Greengrass core device, you can use its token exchange role and role alias instead of creating new ones. Then, you configure your device's AWS IoT thing to use that role and alias.

To create a token exchange IAM role

1. Create an IAM role that your device can use as a token exchange role. Do the following:

   a. Create a file that contains the trust policy document that the token exchange role requires.

      For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

      ```
      nano device-role-trust-policy.json
      ```

      Copy the following JSON into the file.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "credentials.iot.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

b. Create the token exchange role with the trust policy document.

- Replace *GreengrassV2TokenExchangeRole* with the name of the IAM role to create.

```
aws iam create-role --role-name GreengrassV2TokenExchangeRole --assume-role-
policy-document file://device-role-trust-policy.json
```

The response looks similar to the following example, if the request succeeds.

```
{
  "Role": {
    "Path": "/",
    "RoleName": "GreengrassV2TokenExchangeRole",
    "RoleId": "AROAZ2YMUHYHK5OKM77FB",
    "Arn": "arn:aws:iam::123456789012:role/GreengrassV2TokenExchangeRole",
    "CreateDate": "2021-02-06T00:13:29+00:00",
    "AssumeRolePolicyDocument": {
      "Version": "2012-10-17",
      "Statement": [
        {
          "Effect": "Allow",
          "Principal": {
            "Service": "credentials.iot.amazonaws.com"
          },
          "Action": "sts:AssumeRole"
        }
      ]
    }
```

```
  }
```

c. Create a file that contains the access policy document that the token exchange role requires.

For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

```
nano device-role-access-policy.json
```

Copy the following JSON into the file.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:DescribeLogStreams",
        "s3:GetBucketLocation"
      ],
      "Resource": "*"
    }
  ]
}
```

> **Note**
>
> This access policy doesn't allow access to component artifacts in S3 buckets. To deploy custom components that define artifacts in Amazon S3, you must add permissions to the role to allow your core device to retrieve component artifacts. For more information, see Allow access to S3 buckets for component artifacts. If you don't yet have an S3 bucket for component artifacts, you can add these permissions later after you create a bucket.

d. Create the IAM policy from the policy document.

- Replace *GreengrassV2TokenExchangeRoleAccess* with the name of the IAM policy to create.

```
aws iam create-policy --policy-name GreengrassV2TokenExchangeRoleAccess --
policy-document file://device-role-access-policy.json
```

The response looks similar to the following example, if the request succeeds.

```
{
  "Policy": {
    "PolicyName": "GreengrassV2TokenExchangeRoleAccess",
    "PolicyId": "ANPAZ2YMUHYHACI7C5Z66",
    "Arn": "arn:aws:iam::123456789012:policy/
GreengrassV2TokenExchangeRoleAccess",
    "Path": "/",
    "DefaultVersionId": "v1",
    "AttachmentCount": 0,
    "PermissionsBoundaryUsageCount": 0,
    "IsAttachable": true,
    "CreateDate": "2021-02-06T00:37:17+00:00",
    "UpdateDate": "2021-02-06T00:37:17+00:00"
  }
}
```

e. Attach the IAM policy to the token exchange role.

- Replace *GreengrassV2TokenExchangeRole* with the name of the IAM role.
- Replace the policy ARN with the ARN of the IAM policy that you created in the previous step.

```
aws iam attach-role-policy --role-name GreengrassV2TokenExchangeRole --policy-
arn arn:aws:iam::123456789012:policy/GreengrassV2TokenExchangeRoleAccess
```

The command doesn't have any output if the request succeeds.

2. Create an AWS IoT role alias that points to the token exchange role.

- Replace *GreengrassCoreTokenExchangeRoleAlias* with the name of the role alias to create.

- Replace the role ARN with the ARN of the IAM role that you created in the previous step.

```
aws iot create-role-alias --role-alias GreengrassCoreTokenExchangeRoleAlias --role-
arn arn:aws:iam::123456789012:role/GreengrassV2TokenExchangeRole
```

The response looks similar to the following example, if the request succeeds.

```
{
  "roleAlias": "GreengrassCoreTokenExchangeRoleAlias",
  "roleAliasArn": "arn:aws:iot:us-west-2:123456789012:rolealias/
GreengrassCoreTokenExchangeRoleAlias"
}
```

> **Note**
>
> To create a role alias, you must have permission to pass the token exchange IAM role to AWS IoT. If you receive an error message when you try to create a role alias, check that your AWS user has this permission. For more information, see Granting a user permissions to pass a role to an AWS service in the *AWS Identity and Access Management User Guide*.

3. Create and attach an AWS IoT policy that allows your Greengrass core device to use the role alias to assume the token exchange role. If you have set up a Greengrass core device before, you can attach its role alias AWS IoT policy instead of creating a new one. Do the following:

   a. (Optional) Create a file that contains the AWS IoT policy document that the role alias requires.

      For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

      ```
      nano greengrass-v2-iot-role-alias-policy.json
      ```

      Copy the following JSON into the file.

      - Replace the resource ARN with the ARN of your role alias.

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "iot:AssumeRoleWithCertificate",
      "Resource": "arn:aws:iot:us-west-2:123456789012:rolealias/
GreengrassCoreTokenExchangeRoleAlias"
    }
  ]
}
```

b. Create an AWS IoT policy from the policy document.

- Replace *GreengrassCoreTokenExchangeRoleAliasPolicy* with the name of the AWS IoT policy to create.

```
aws iot create-policy --policy-name GreengrassCoreTokenExchangeRoleAliasPolicy
 --policy-document file://greengrass-v2-iot-role-alias-policy.json
```

The response looks similar to the following example, if the request succeeds.

```
{
  "policyName": "GreengrassCoreTokenExchangeRoleAliasPolicy",
  "policyArn": "arn:aws:iot:us-west-2:123456789012:policy/
GreengrassCoreTokenExchangeRoleAliasPolicy",
  "policyDocument": "{
    \\"Version\\":\\"2012-10-17       \\",
    \\"Statement\\": [
      {
        \\"Effect\\": \\"Allow\\",
        \\"Action\\": \\"iot:AssumeRoleWithCertificate\\",
        \\"Resource\\": \\"arn:aws:iot:us-west-2:123456789012:rolealias/
GreengrassCoreTokenExchangeRoleAlias\\"
      }
    ]
  }",
  "policyVersionId": "1"
}
```

c. Attach the AWS IoT policy to the AWS IoT thing's certificate.

- Replace *GreengrassCoreTokenExchangeRoleAliasPolicy* with the name of the role alias AWS IoT policy.
- Replace the target ARN with the ARN of the certificate for your AWS IoT thing.

```
aws iot attach-policy --policy-name GreengrassCoreTokenExchangeRoleAliasPolicy
 --target arn:aws:iot:us-west-2:123456789012:cert/
aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4
```

The command doesn't have any output if the request succeeds.

Download certificates to the device

Earlier, you downloaded your device's certificate to your development computer. In this section, you copy the certificate to your core device to set up the device with the certificates that it uses to connect to AWS IoT. You also download the Amazon root certificate authority (CA) certificate. If you use an HSM, you also import the certificate file into the HSM in this section.

- If you created the thing certificate and private key in the AWS IoT service earlier, follow the steps to download the certificates with private key and certificate files.
- If you created the thing certificate from a private key in a hardware security module (HSM) earlier, follow the steps to download the certificates with the private key and certificate in an HSM.

Download certificates with private key and certificate files

To download certificates to the device

1. Copy the AWS IoT thing certificate from your development computer to the device. If SSH and SCP are enabled on the development computer and the device, you can use the `scp` command on your development computer to transfer the certificate. Replace *device-ip-address* with the IP address of your device.

```
scp -r greengrass-v2-certs/ device-ip-address:~
```

2. Create the Greengrass root folder on the device. You'll later install the AWS IoT Greengrass Core software to this folder.

   > **Note**
   >
   > Windows has a path length limitation of 260 characters. If you are using Windows, use a root folder like `C:\greengrass\v2` or `D:\greengrass\v2` to keep the Greengrass components paths under the 260 character limit.

   Linux or Unix

   - Replace */greengrass/v2* with the folder to use.

   ```
   sudo mkdir -p /greengrass/v2
   ```

   Windows Command Prompt

   - Replace *C:\greengrass\v2* with the folder to use.

   ```
   mkdir C:\greengrass\v2
   ```

   PowerShell

   - Replace *C:\greengrass\v2* with the folder to use.

   ```
   mkdir C:\greengrass\v2
   ```

3. (Linux only) Set the permissions of the parent of the Greengrass root folder.

   - Replace */greengrass* with the parent of the root folder.

   ```
   sudo chmod 755 /greengrass
   ```

4. Copy the AWS IoT thing certificates to the Greengrass root folder.

Linux or Unix

- Replace */greengrass/v2* with the Greengrass root folder.

```
sudo cp -R ~/greengrass-v2-certs/* /greengrass/v2
```

Windows Command Prompt

- Replace *C:\greengrass\v2* with the folder to use.

```
robocopy %USERPROFILE%\greengrass-v2-certs C:\greengrass\v2 /E
```

PowerShell

- Replace *C:\greengrass\v2* with the folder to use.

```
cp -Path ~\greengrass-v2-certs\* -Destination C:\greengrass\v2
```

5. Download the Amazon root certificate authority (CA) certificate. AWS IoT certificates are associated with Amazon's root CA certificate by default.

   Linux or Unix

   ```
   sudo curl -o /greengrass/v2/AmazonRootCA1.pem https://www.amazontrust.com/
   repository/AmazonRootCA1.pem
   ```

   Windows Command Prompt (CMD)

   ```
   curl -o C:\greengrass\v2\\AmazonRootCA1.pem https://www.amazontrust.com/
   repository/AmazonRootCA1.pem
   ```

   PowerShell

   ```
   iwr -Uri https://www.amazontrust.com/repository/AmazonRootCA1.pem -OutFile C:
   \greengrass\v2\\AmazonRootCA1.pem
   ```

Download certificates with the private key and certificate in an HSM

> **Note**
>
> This feature is available for v2.5.3 and later of the Greengrass nucleus component. AWS IoT Greengrass doesn't currently support this feature on Windows core devices.

To download certificates to the device

1. Copy the AWS IoT thing certificate from your development computer to the device. If SSH and SCP are enabled on the development computer and the device, you can use the `scp` command on your development computer to transfer the certificate. Replace *`device-ip-address`* with the IP address of your device.

   ```
   scp -r greengrass-v2-certs/ device-ip-address:~
   ```

2. Create the Greengrass root folder on the device. You'll later install the AWS IoT Greengrass Core software to this folder.

   > **Note**
   >
   > Windows has a path length limitation of 260 characters. If you are using Windows, use a root folder like `C:\greengrass\v2` or `D:\greengrass\v2` to keep the Greengrass components paths under the 260 character limit.

   Linux or Unix

   - Replace */greengrass/v2* with the folder to use.

   ```
   sudo mkdir -p /greengrass/v2
   ```

   Windows Command Prompt

   - Replace *C:\greengrass\v2* with the folder to use.

```
mkdir C:\greengrass\v2
```

PowerShell

- Replace *C:\greengrass\v2* with the folder to use.

```
mkdir C:\greengrass\v2
```

3. (Linux only) Set the permissions of the parent of the Greengrass root folder.

   - Replace */greengrass* with the parent of the root folder.

   ```
   sudo chmod 755 /greengrass
   ```

4. Import the thing certificate file, `~/greengrass-v2-certs/device.pem.crt`, into the HSM. Check the documentation for your HSM to learn how to import certificates into it. Import the certificate using the same token, slot ID, user PIN, object label, and object ID (if your HSM uses one) where you generated the private key in the HSM earlier.

   > **Note**
   >
   > If you generated the private key earlier without an object ID, and the certificate has an object ID, set the private key's object ID to the same value as the certificate. Check the documentation for your HSM to learn how to set the object ID for the private key object.

5. (Optional) Delete the thing certificate file, so that it exists only in the HSM.

   ```
   rm ~/greengrass-v2-certs/device.pem.crt
   ```

6. Download the Amazon root certificate authority (CA) certificate. AWS IoT certificates are associated with Amazon's root CA certificate by default.

Linux or Unix

```
sudo curl -o /greengrass/v2/AmazonRootCA1.pem https://www.amazontrust.com/
repository/AmazonRootCA1.pem
```

Windows Command Prompt (CMD)

```
curl -o C:\greengrass\v2\\AmazonRootCA1.pem https://www.amazontrust.com/
repository/AmazonRootCA1.pem
```

PowerShell

```
iwr -Uri https://www.amazontrust.com/repository/AmazonRootCA1.pem -OutFile C:
\greengrass\v2\\AmazonRootCA1.pem
```

Set up the device environment

Follow the steps in this section to set up a Linux or Windows device to use as your AWS IoT Greengrass core device.

Set up a Linux device

To set up a Linux device for AWS IoT Greengrass V2

1. Install the Java runtime, which AWS IoT Greengrass Core software requires to run. We recommend that you use Amazon Corretto or OpenJDK long-term support versions. Version 8 or higher is required. The following commands show you how to install OpenJDK on your device.

   - For Debian-based or Ubuntu-based distributions:

     ```
     sudo apt install default-jdk
     ```

   - For Red Hat-based distributions:

     ```
     sudo yum install java-11-openjdk-devel
     ```

   - For Amazon Linux 2:

```
sudo amazon-linux-extras install java-openjdk11
```

- For Amazon Linux 2023:

```
sudo dnf install java-11-amazon-corretto -y
```

When the installation completes, run the following command to verify that Java runs on your Linux device.

```
java -version
```

The command prints the version of Java that runs on the device. For example, on a Debian-based distribution, the output might look similar to the following sample.

```
openjdk version "11.0.9.1" 2020-11-04
OpenJDK Runtime Environment (build 11.0.9.1+1-post-Debian-1deb10u2)
OpenJDK 64-Bit Server VM (build 11.0.9.1+1-post-Debian-1deb10u2, mixed mode)
```

2. (Optional) Create the default system user and group that runs components on the device. You can also choose to let the AWS IoT Greengrass Core software installer create this user and group during installation with the `--component-default-user` installer argument. For more information, see Installer arguments.

```
sudo useradd --system --create-home ggc_user
sudo groupadd --system ggc_group
```

3. Verify that the user that runs the AWS IoT Greengrass Core software (typically `root`), has permission to run `sudo` with any user and any group.

   a. Run the following command to open the `/etc/sudoers` file.

   ```
   sudo visudo
   ```

   b. Verify that the permission for the user looks like the following example.

   ```
   root    ALL=(ALL:ALL) ALL
   ```

4. (Optional) To run containerized Lambda functions, you must enable cgroups v1, and you must enable and mount the *memory* and *devices* cgroups. If you don't plan to run containerized Lambda functions, you can skip this step.

   To enable these cgroups options, boot the device with the following Linux kernel parameters.

   ```
   cgroup_enable=memory cgroup_memory=1 systemd.unified_cgroup_hierarchy=0
   ```

   For information about viewing and setting kernel parameters for your device, see the documentation for your operating system and boot loader. Follow the instructions to permanently set the kernel parameters.
5. Install all other required dependencies on your device as indicated by the list of requirements in Device requirements.

Set up a Windows device

> **Note**
>
> This feature is available for v2.5.0 and later of the Greengrass nucleus component.

To set up a Windows device for AWS IoT Greengrass V2

1. Install the Java runtime, which AWS IoT Greengrass Core software requires to run. We recommend that you use Amazon Corretto or OpenJDK long-term support versions. Version 8 or higher is required.
2. Check whether Java is available on the PATH system variable, and add it if not. The LocalSystem account runs the AWS IoT Greengrass Core software, so you must add Java to the PATH system variable instead of the PATH user variable for your user. Do the following:

   a. Press the Windows key to open the start menu.
   b. Type **`environment variables`** to search for the system options from the start menu.
   c. In the start menu search results, choose **Edit the system environment variables** to open the **System properties** window.
   d. Choose **Environment variables...** to open the **Environment Variables** window.
   e. Under **System variables**, select **Path**, and then choose **Edit**. In the **Edit environment variable** window, you can view each path on a separate line.

f. Check if the path to the Java installation's `bin` folder is present. The path might look similar to the following example.

```
C:\\Program Files\\Amazon Corretto\\jdk11.0.13_8\\bin
```

g. If the Java installation's `bin` folder is missing from **Path**, choose **New** to add it, and then choose **OK**.

3. Open the Windows Command Prompt (`cmd.exe`) as an administrator.
4. Create the default user in the LocalSystem account on the Windows device. Replace *`password`* with a secure password.

```
net user /add ggc_user password
```

> **Tip**
>
> Depending on your Windows configuration, the user's password might be set to expire at a date in the future. To ensure your Greengrass applications continue to operate, track when the password expires, and update it before it expires. You can also set the user's password to never expire.
>
> - To check when a user and its password expire, run the following command.
>
>   ```
>   net user ggc_user | findstr /C:expires
>   ```
>
> - To set a user's password to never expire, run the following command.
>
>   ```
>   wmic UserAccount where "Name='ggc_user'" set PasswordExpires=False
>   ```
>
> - If you're using Windows 10 or later where the wmic command is deprecated, run the following PowerShell command.
>
>   ```
>   Get-CimInstance -Query "SELECT * from Win32_UserAccount WHERE name =
>    'ggc_user'" | Set-CimInstance -Property @{PasswordExpires="False"}
>   ```

5. Download and install the PsExec utility from Microsoft on the device.

6. Use the PsExec utility to store the user name and password for the default user in the Credential Manager instance for the LocalSystem account. Replace *password* with the user's password that you set earlier.

   ```
   psexec -s cmd /c cmdkey /generic:ggc_user /user:ggc_user /pass:password
   ```

   If the **PsExec License Agreement** opens, choose **Accept** to agree to the license and run the command.

   > **Note**
   >
   > On Windows devices, the LocalSystem account runs the Greengrass nucleus, and you must use the PsExec utility to store the default user information in the LocalSystem account. Using the Credential Manager application stores this information in the Windows account of the currently logged on user, instead of the LocalSystem account.

Download the AWS IoT Greengrass Core software

You can download the latest version of the AWS IoT Greengrass Core software from the following location:

- https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip

> **Note**
>
> You can download a specific version of the AWS IoT Greengrass Core software from the following location. Replace *version* with the version to download.
>
> ```
> https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-version.zip
> ```

To download the AWS IoT Greengrass Core software

1. On your core device, download the AWS IoT Greengrass Core software to a file named `greengrass-nucleus-latest.zip`.

Linux or Unix

```
curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-
latest.zip > greengrass-nucleus-latest.zip
```

Windows Command Prompt (CMD)

```
curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-
latest.zip > greengrass-nucleus-latest.zip
```

PowerShell

```
iwr -Uri https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-
latest.zip -OutFile greengrass-nucleus-latest.zip
```

By downloading this software, you agree to the Greengrass Core Software License Agreement.

2. (Optional) To verify the Greengrass nucleus software signature

> **Note**
>
> This feature is available with Greengrass nucleus version 2.9.5 and later.

a. Use the following command to verify your Greengrass nucleus artifact's signature:

Linux or Unix

```
jarsigner -verify -certs -verbose greengrass-nucleus-latest.zip
```

Windows Command Prompt (CMD)

The file name might look different depending on the JDK version you install. Replace *jdk17.0.6_10* with the JDK version you installed.

```
"C:\\Program Files\\Amazon Corretto\\jdk17.0.6_10\\bin\\jarsigner.exe" -
verify -certs -verbose greengrass-nucleus-latest.zip
```

PowerShell

The file name might look different depending on the JDK version you install. Replace *jdk17.0.6_10* with the JDK version you installed.

```
'C:\\Program Files\\Amazon Corretto\\jdk17.0.6_10\\bin\\jarsigner.exe' -
verify -certs -verbose greengrass-nucleus-latest.zip
```

b. The `jarsigner` invocation yields output that indicates the results of the verification.

   i. If the Greengrass nucleus zip file is signed, the output contains the following statement:

   ```
   jar verified.
   ```

   ii. If the Greengrass nucleus zip file isn't signed, the output contains the following statement:

   ```
   jar is unsigned.
   ```

c. If you provided the Jarsigner `-certs` option along with `-verify` and `-verbose` options, the output also includes detailed signer certificate information.

3. Unzip the AWS IoT Greengrass Core software to a folder on your device. Replace *GreengrassInstaller* with the folder that you want to use.

   Linux or Unix

   ```
   unzip greengrass-nucleus-latest.zip -d GreengrassInstaller && rm greengrass-
   nucleus-latest.zip
   ```

   Windows Command Prompt (CMD)

   ```
   mkdir GreengrassInstaller && tar -xf greengrass-nucleus-latest.zip -
   C GreengrassInstaller && del greengrass-nucleus-latest.zip
   ```

   PowerShell

   ```
   Expand-Archive -Path greengrass-nucleus-latest.zip -DestinationPath .\
   \GreengrassInstaller
   ```

```
rm greengrass-nucleus-latest.zip
```

4. (Optional) Run the following command to see the version of the AWS IoT Greengrass Core software.

```
java -jar ./GreengrassInstaller/lib/Greengrass.jar --version
```

> **Important**
>
> If you install a version of the Greengrass nucleus earlier than v2.4.0, don't remove this folder after you install the AWS IoT Greengrass Core software. The AWS IoT Greengrass Core software uses the files in this folder to run.
> If you downloaded the latest version of the software, you install v2.4.0 or later, and you can remove this folder after you install the AWS IoT Greengrass Core software.

Install the AWS IoT Greengrass Core software

Run the installer with arguments that specify the following actions:

- Install from a partial configuration file that specifies to use the AWS resources and certificates that you created earlier. The AWS IoT Greengrass Core software uses a configuration file that specifies the configuration of every Greengrass component on the device. The installer creates a complete configuration file from the partial configuration file that you provide.
- Specify to use the `ggc_user` system user to run software components on the core device. On Linux devices, this command also specifies to use the `ggc_group` system group, and the installer creates the system user and group for you.
- Set up the AWS IoT Greengrass Core software as a system service that runs at boot. On Linux devices, this requires the Systemd init system.

> **Important**
>
> On Windows core devices, you must set up the AWS IoT Greengrass Core software as a system service.

For more information about the arguments that you can specify, see Installer arguments.

> **Note**
>
> If you are running AWS IoT Greengrass on a device with limited memory, you can control the amount of memory that AWS IoT Greengrass Core software uses. To control memory allocation, you can set JVM heap size options in the `jvmOptions` configuration parameter in your nucleus component. For more information, see Control memory allocation with JVM options.

- If you created the thing certificate and private key in the AWS IoT service earlier, follow the steps to install the AWS IoT Greengrass Core software with private key and certificate files.
- If you created the thing certificate from a private key in a hardware security module (HSM) earlier, follow the steps to install the AWS IoT Greengrass Core software with the private key and certificate in an HSM.

Install the AWS IoT Greengrass Core software with private key and certificate files

To install the AWS IoT Greengrass Core software

1. Check the version of the AWS IoT Greengrass Core software.

   - Replace *GreengrassInstaller* with the path to the folder that contains the software.

   ```
   java -jar ./GreengrassInstaller/lib/Greengrass.jar --version
   ```

2. Use a text editor to create a configuration file named `config.yaml` to provide to the installer.

   For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

   ```
   nano GreengrassInstaller/config.yaml
   ```

   Copy the following YAML content into the file. This partial configuration file specifies system parameters and Greengrass nucleus parameters.

   ```
   ---
   system:
     certificateFilePath: "/greengrass/v2/device.pem.crt"
   ```

```
  privateKeyPath: "/greengrass/v2/private.pem.key"
  rootCaPath: "/greengrass/v2/AmazonRootCA1.pem"
  rootpath: "/greengrass/v2"
  thingName: "MyGreengrassCore"
services:
  aws.greengrass.Nucleus:
    componentType: "NUCLEUS"
    version: "2.18.4"
    configuration:
      awsRegion: "us-west-2"
      iotRoleAlias: "GreengrassCoreTokenExchangeRoleAlias"
      iotDataEndpoint: "device-data-prefix-ats.iot.us-west-2.amazonaws.com"
      iotCredEndpoint: "device-credentials-prefix.credentials.iot.us-
west-2.amazonaws.com"
```

Then, do the following:

- Replace each instance of */greengrass/v2* with the Greengrass root folder.
- Replace *MyGreengrassCore* with the name of the AWS IoT thing.
- Replace *2.18.4* with the version of the AWS IoT Greengrass Core software.
- Replace *us-west-2* with the AWS Region where you created the resources.
- Replace *GreengrassCoreTokenExchangeRoleAlias* with the name of the token exchange role alias.
- Replace the `iotDataEndpoint` with your AWS IoT data endpoint.
- Replace the `iotCredEndpoint` with your AWS IoT credentials endpoint.

Note

In this configuration file, you can customize other nucleus configuration options such as the ports and network proxy to use, as shown in the following example. For more information, see Greengrass nucleus configuration.

```
---
system:
  certificateFilePath: "/greengrass/v2/device.pem.crt"
  privateKeyPath: "/greengrass/v2/private.pem.key"
  rootCaPath: "/greengrass/v2/AmazonRootCA1.pem"
  rootpath: "/greengrass/v2"
  thingName: "MyGreengrassCore"
```

```
services:
  aws.greengrass.Nucleus:
    componentType: "NUCLEUS"
    version: "2.18.4"
    configuration:
      awsRegion: "us-west-2"
      iotRoleAlias: "GreengrassCoreTokenExchangeRoleAlias"
      iotCredEndpoint: "device-credentials-prefix.credentials.iot.us-
west-2.amazonaws.com"
      iotDataEndpoint: "device-data-prefix-ats.iot.us-west-2.amazonaws.com"
      mqtt:
        port: 443
      greengrassDataPlanePort: 443
      networkProxy:
        noProxyAddresses: "http://192.168.0.1,www.example.com"
        proxy:
          url: "https://my-proxy-server:1100"
          username: "Mary_Major"
          password: "pass@word1357"
```

3. Run the installer, and specify `--init-config` to provide the configuration file.

   - Replace */greengrass/v2* or *C:\greengrass\v2* with the Greengrass root folder.
   - Replace each instance of *GreengrassInstaller* with the folder where you unpacked the installer.

   Linux or Unix

   ```
   sudo -E java -Droot="/greengrass/v2" -Dlog.store=FILE \
     -jar ./GreengrassInstaller/lib/Greengrass.jar \
     --init-config ./GreengrassInstaller/config.yaml \
     --component-default-user ggc_user:ggc_group \
     --setup-system-service true
   ```

   Windows Command Prompt (CMD)

   ```
   java -Droot="C:\greengrass\v2" "-Dlog.store=FILE" ^
     -jar ./GreengrassInstaller/lib/Greengrass.jar ^
     --init-config ./GreengrassInstaller/config.yaml ^
     --component-default-user ggc_user ^
   ```

```
  --setup-system-service true
```

PowerShell

```
java -Droot="C:\greengrass\v2" "-Dlog.store=FILE" `
  -jar ./GreengrassInstaller/lib/Greengrass.jar `
  --init-config ./GreengrassInstaller/config.yaml `
  --component-default-user ggc_user `
  --setup-system-service true
```

> **Important**
>
> On Windows core devices, you must specify `--setup-system-service true` to set up the AWS IoT Greengrass Core software as a system service.

If you specify `--setup-system-service true`, the installer prints `Successfully set up Nucleus as a system service` if it set up and ran the software as a system service. Otherwise, the installer doesn't output any message if it installs the software successfully.

> **Note**
>
> You can't use the `deploy-dev-tools` argument to deploy local development tools when you run the installer without the `--provision true` argument. For information about deploying the Greengrass CLI directly on your device, see Greengrass Command Line Interface.

4. Verify the installation by viewing the files in the root folder.

   Linux or Unix

   ```
   ls /greengrass/v2
   ```

   Windows Command Prompt (CMD)

   ```
   dir C:\greengrass\v2
   ```

PowerShell

```
ls C:\greengrass\v2
```

If the installation succeeded, the root folder contains several folders, such as `config`, `packages`, and `logs`.

Install the AWS IoT Greengrass Core software with the private key and certificate in an HSM

> **Note**
>
> This feature is available for v2.5.3 and later of the Greengrass nucleus component. AWS IoT Greengrass doesn't currently support this feature on Windows core devices.

To install the AWS IoT Greengrass Core software

1. Check the version of the AWS IoT Greengrass Core software.

   - Replace *GreengrassInstaller* with the path to the folder that contains the software.

   ```
   java -jar ./GreengrassInstaller/lib/Greengrass.jar --version
   ```

2. To enable the AWS IoT Greengrass Core software to use the private key and certificate in the HSM, install the PKCS#11 provider component when you install the AWS IoT Greengrass Core software. The PKCS#11 provider component is a plugin that you can configure during installation. You can download the latest version of the PKCS#11 provider component from the following location:

   - https://d2s8p88vqu9w66.cloudfront.net/releases/Pkcs11Provider/aws.greengrass.crypto.Pkcs11Provider-latest.jar

   Download the PKCS#11 provider plugin to a file named `aws.greengrass.crypto.Pkcs11Provider.jar`. Replace *GreengrassInstaller* with the folder that you want to use.

```
curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/Pkcs11Provider/
aws.greengrass.crypto.Pkcs11Provider-latest.jar > GreengrassInstaller/
aws.greengrass.crypto.Pkcs11Provider.jar
```

By downloading this software, you agree to the Greengrass Core Software License Agreement.

3. Use a text editor to create a configuration file named `config.yaml` to provide to the installer.

   For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

   ```
   nano GreengrassInstaller/config.yaml
   ```

   Copy the following YAML content into the file. This partial configuration file specifies system parameters, Greengrass nucleus parameters, and PKCS#11 provider parameters.

   ```
   ---
   system:
     certificateFilePath: "pkcs11:object=iotdevicekey;type=cert"
     privateKeyPath: "pkcs11:object=iotdevicekey;type=private"
     rootCaPath: "/greengrass/v2/AmazonRootCA1.pem"
     rootpath: "/greengrass/v2"
     thingName: "MyGreengrassCore"
   services:
     aws.greengrass.Nucleus:
       componentType: "NUCLEUS"
       version: "2.18.4"
       configuration:
         awsRegion: "us-west-2"
         iotRoleAlias: "GreengrassCoreTokenExchangeRoleAlias"
         iotDataEndpoint: "device-data-prefix-ats.iot.us-west-2.amazonaws.com"
         iotCredEndpoint: "device-credentials-prefix.credentials.iot.us-
   west-2.amazonaws.com"
     aws.greengrass.crypto.Pkcs11Provider:
       configuration:
         name: "softhsm_pkcs11"
         library: "/usr/local/Cellar/softhsm/2.6.1/lib/softhsm/libsofthsm2.so"
         slot: 1
         userPin: "1234"
   ```

   Then, do the following:

- Replace each instance of *iotdevicekey* in the PKCS#11 URIs with the object label where you created the private key and imported the certificate.
- Replace each instance of */greengrass/v2* with the Greengrass root folder.
- Replace *MyGreengrassCore* with the name of the AWS IoT thing.
- Replace *2.18.4* with the version of the AWS IoT Greengrass Core software.
- Replace *us-west-2* with the AWS Region where you created the resources.
- Replace *GreengrassCoreTokenExchangeRoleAlias* with the name of the token exchange role alias.
- Replace the `iotDataEndpoint` with your AWS IoT data endpoint.
- Replace the `iotCredEndpoint` with your AWS IoT credentials endpoint.
- Replace the configuration parameters for the `aws.greengrass.crypto.Pkcs11Provider` component with the values for the HSM configuration on the core device.

Note

In this configuration file, you can customize other nucleus configuration options such as the ports and network proxy to use, as shown in the following example. For more information, see Greengrass nucleus configuration.

```
---
system:
  certificateFilePath: "pkcs11:object=iotdevicekey;type=cert"
  privateKeyPath: "pkcs11:object=iotdevicekey;type=private"
  rootCaPath: "/greengrass/v2/AmazonRootCA1.pem"
  rootpath: "/greengrass/v2"
  thingName: "MyGreengrassCore"
services:
  aws.greengrass.Nucleus:
    componentType: "NUCLEUS"
    version: "2.18.4"
    configuration:
      awsRegion: "us-west-2"
      iotRoleAlias: "GreengrassCoreTokenExchangeRoleAlias"
      iotDataEndpoint: "device-data-prefix-ats.iot.us-west-2.amazonaws.com"
      iotCredEndpoint: "device-credentials-prefix.credentials.iot.us-
west-2.amazonaws.com"
```

```
      mqtt:
        port: 443
      greengrassDataPlanePort: 443
      networkProxy:
        noProxyAddresses: "http://192.168.0.1,www.example.com"
        proxy:
          url: "https://my-proxy-server:1100"
          username: "Mary_Major"
          password: "pass@word1357"
  aws.greengrass.crypto.Pkcs11Provider:
    configuration:
      name: "softhsm_pkcs11"
      library: "/usr/local/Cellar/softhsm/2.6.1/lib/softhsm/libsofthsm2.so"
      slot: 1
      userPin: "1234"
```

4. Run the installer, and specify `--init-config` to provide the configuration file.

   - Replace */greengrass/v2* with the Greengrass root folder.
   - Replace each instance of *GreengrassInstaller* with the folder where you unpacked the installer.

```
sudo -E java -Droot="/greengrass/v2" -Dlog.store=FILE \
  -jar ./GreengrassInstaller/lib/Greengrass.jar \
  --trusted-plugin ./GreengrassInstaller/aws.greengrass.crypto.Pkcs11Provider.jar \
  --init-config ./GreengrassInstaller/config.yaml \
  --component-default-user ggc_user:ggc_group \
  --setup-system-service true
```

> **Important**
>
> On Windows core devices, you must specify `--setup-system-service true` to set up the AWS IoT Greengrass Core software as a system service.

If you specify `--setup-system-service true`, the installer prints `Successfully set up Nucleus as a system service` if it set up and ran the software as a system service. Otherwise, the installer doesn't output any message if it installs the software successfully.

> **Note**
>
> You can't use the `deploy-dev-tools` argument to deploy local development tools when you run the installer without the `--provision true` argument. For information about deploying the Greengrass CLI directly on your device, see Greengrass Command Line Interface.

5. Verify the installation by viewing the files in the root folder.

   Linux or Unix

   ```
   ls /greengrass/v2
   ```

   Windows Command Prompt (CMD)

   ```
   dir C:\greengrass\v2
   ```

   PowerShell

   ```
   ls C:\greengrass\v2
   ```

   If the installation succeeded, the root folder contains several folders, such as `config`, `packages`, and `logs`.

If you installed the AWS IoT Greengrass Core software as a system service, the installer runs the software for you. Otherwise, you must run the software manually. For more information, see Run the AWS IoT Greengrass Core software.

> **Note**
>
> When the AWS IoT Greengrass Core software connects to the cloud, your device will be recognized as a Core device.

For more information about how to configure and use the software and AWS IoT Greengrass, see the following:

- Configure the AWS IoT Greengrass Core software
- Develop AWS IoT Greengrass components
- Deploy AWS IoT Greengrass components to devices
- Greengrass Command Line Interface

Install AWS IoT Greengrass Core software with AWS IoT fleet provisioning

This feature is available for v2.4.0 and later of the Greengrass nucleus component.

With AWS IoT fleet provisioning, you can configure AWS IoT to generate and securely deliver X.509 device certificates and private keys to your devices when they connect to AWS IoT for the first time. AWS IoT provides client certificates that are signed by the Amazon Root certificate authority (CA). You can also configure AWS IoT to specify thing groups, thing types, and permissions for Greengrass core devices that you provision with fleet provisioning. You define a *provisioning template* to define how AWS IoT provisions each device. The provisioning template specifies the thing, policy, and certificate resources to create for a device when provisioning. For more information, see Provisioning templates in the *AWS IoT Core Developer Guide*.

AWS IoT Greengrass provides an AWS IoT fleet provisioning plugin that you can use to install the AWS IoT Greengrass Core software using AWS resources created by AWS IoT fleet provisioning. The fleet provisioning plugin uses *provisioning by claim*. Devices use a provisioning claim certificate and private key to obtain a unique X.509 device certificate and private key that they can use for regular operations. You can embed the claim certificate and private key in each device during manufacturing, so your customers can activate devices later when each device comes online. You can use the same claim certificate and private key for multiple devices. For more information, see Provisioning by claim in the *AWS IoT Core Developer Guide*.

> **Note**
>
> The fleet provisioning plugin for Greengrass nucleus doesn't currently support storing private key and certificate files in a hardware security module (HSM). To use an HSM with Greengrass nucleus, install the AWS IoT Greengrass Core software with manual provisioning. Greengrass nucleus lite v2.5.0 and later support TPM with fleet provisioning natively.

To install the AWS IoT Greengrass Core software with AWS IoT fleet provisioning, you must set up resources in your AWS account that AWS IoT uses to provision Greengrass core devices. These resources include a provisioning template, claim certificates, and a token exchange IAM role. After you create these resources, you can reuse them to provision multiple core devices in a fleet. For more information, see Set up AWS IoT fleet provisioning for Greengrass core devices.

> **Important**
>
> Before you download the AWS IoT Greengrass Core software, check that your core device meets the requirements to install and run the AWS IoT Greengrass Core software v2.0.

Topics

- Prerequisites
- Retrieve AWS IoT endpoints
- Download certificates to the device
- Set up the device environment
- Download the AWS IoT Greengrass Core software
- Download the AWS IoT fleet provisioning plugin
- Install the AWS IoT Greengrass Core software
- Set up AWS IoT fleet provisioning for Greengrass core devices
- Configure the AWS IoT fleet provisioning plugin
- AWS IoT fleet provisioning plugin changelog

Prerequisites

To install the AWS IoT Greengrass Core software with AWS IoT fleet provisioning, you must first set up AWS IoT fleet provisioning for Greengrass core devices. After you complete these steps once, you can use fleet provisioning to install the AWS IoT Greengrass Core software on any number of devices.

Retrieve AWS IoT endpoints

Get the AWS IoT endpoints for your AWS account, and save them to use later. Your device uses these endpoints to connect to AWS IoT. Do the following:

1. Get the AWS IoT data endpoint for your AWS account.

   ```
   aws iot describe-endpoint --endpoint-type iot:Data-ATS
   ```

   The response looks similar to the following example, if the request succeeds.

   ```
   {
     "endpointAddress": "device-data-prefix-ats.iot.us-west-2.amazonaws.com"
   }
   ```

2. Get the AWS IoT credentials endpoint for your AWS account.

   ```
   aws iot describe-endpoint --endpoint-type iot:CredentialProvider
   ```

   The response looks similar to the following example, if the request succeeds.

   ```
   {
     "endpointAddress": "device-credentials-prefix.credentials.iot.us-
   west-2.amazonaws.com"
   }
   ```

Download certificates to the device

The device uses a claim certificate and private key to authenticate its request to provision AWS resources and acquire an X.509 device certificate. You can embed the claim certificate and private key into the device during manufacturing, or copy the certificate and key to the device during installation. In this section, you copy the claim certificate and private key to the device. You also download the Amazon Root certificate authority (CA) certificate to the device.

> **Important**
>
> Provisioning claim private keys should be secured at all times, including on Greengrass core devices. We recommend that you use Amazon CloudWatch metrics and logs to monitor for indications of misuse, such as unauthorized use of the claim certificate to provision devices. If you detect misuse, disable the provisioning claim certificate so that it can't be used for device provisioning. For more information, see Monitoring AWS IoT in the *AWS IoT Core Developer Guide*.

To help you better manage the number of devices, and which devices, that register themselves in your AWS account, you can specify a pre-provisioning hook when you create a fleet provisioning template. A pre-provisioning hook is an AWS Lambda function that validates template parameters that devices provide during registration. For example, you might create a pre-provisioning hook that checks a device ID against a database to verify that the device has permission to provision. For more information, see Pre-provisioning hooks in the *AWS IoT Core Developer Guide*.

To download claim certificates to the device

1. Copy the claim certificate and private key to the device. If SSH and SCP are enabled on the development computer and the device, you can use the `scp` command on your development computer to transfer the claim certificate and private key. The following example command transfers these files a folder named `claim-certs` on your development computer to the device. Replace *`device-ip-address`* with the IP address of your device.

   ```
   scp -r claim-certs/ device-ip-address:~
   ```

2. Create the Greengrass root folder on the device. You'll later install the AWS IoT Greengrass Core software to this folder.

   Note

   Windows has a path length limitation of 260 characters. If you are using Windows, use a root folder like `C:\greengrass\v2` or `D:\greengrass\v2` to keep the Greengrass components paths under the 260 character limit.

   Linux or Unix

   - Replace */greengrass/v2* with the folder to use.

   ```
   sudo mkdir -p /greengrass/v2
   ```

   Windows Command Prompt

   - Replace *C:\greengrass\v2* with the folder to use.

```
mkdir C:\greengrass\v2
```

PowerShell

- Replace *C:\greengrass\v2* with the folder to use.

```
mkdir C:\greengrass\v2
```

3. (Linux only) Set the permissions of the parent of the Greengrass root folder.

   - Replace */greengrass* with the parent of the root folder.

   ```
   sudo chmod 755 /greengrass
   ```

4. Move the claim certificates to the Greengrass root folder.

   - Replace */greengrass/v2* or *C:\greengrass\v2* with the Greengrass root folder.

   Linux or Unix

   ```
   sudo mv ~/claim-certs /greengrass/v2
   ```

   Windows Command Prompt (CMD)

   ```
   move %USERPROFILE%\claim-certs C:\greengrass\v2
   ```

   PowerShell

   ```
   mv -Path ~\claim-certs -Destination C:\greengrass\v2
   ```

5. Download the Amazon root certificate authority (CA) certificate. AWS IoT certificates are associated with Amazon's root CA certificate by default.

Linux or Unix

```
sudo curl -o /greengrass/v2/AmazonRootCA1.pem https://www.amazontrust.com/
repository/AmazonRootCA1.pem
```

Windows Command Prompt (CMD)

```
curl -o C:\greengrass\v2\\AmazonRootCA1.pem https://www.amazontrust.com/
repository/AmazonRootCA1.pem
```

PowerShell

```
iwr -Uri https://www.amazontrust.com/repository/AmazonRootCA1.pem -OutFile C:
\greengrass\v2\\AmazonRootCA1.pem
```

Set up the device environment

Follow the steps in this section to set up a Linux or Windows device to use as your AWS IoT Greengrass core device.

Set up a Linux device

To set up a Linux device for AWS IoT Greengrass V2

1. Install the Java runtime, which AWS IoT Greengrass Core software requires to run. We recommend that you use Amazon Corretto or OpenJDK long-term support versions. Version 8 or higher is required. The following commands show you how to install OpenJDK on your device.

   - For Debian-based or Ubuntu-based distributions:

     ```
     sudo apt install default-jdk
     ```

   - For Red Hat-based distributions:

     ```
     sudo yum install java-11-openjdk-devel
     ```

   - For Amazon Linux 2:

```
sudo amazon-linux-extras install java-openjdk11
```

- For Amazon Linux 2023:

```
sudo dnf install java-11-amazon-corretto -y
```

When the installation completes, run the following command to verify that Java runs on your Linux device.

```
java -version
```

The command prints the version of Java that runs on the device. For example, on a Debian-based distribution, the output might look similar to the following sample.

```
openjdk version "11.0.9.1" 2020-11-04
OpenJDK Runtime Environment (build 11.0.9.1+1-post-Debian-1deb10u2)
OpenJDK 64-Bit Server VM (build 11.0.9.1+1-post-Debian-1deb10u2, mixed mode)
```

2. (Optional) Create the default system user and group that runs components on the device. You can also choose to let the AWS IoT Greengrass Core software installer create this user and group during installation with the `--component-default-user` installer argument. For more information, see Installer arguments.

```
sudo useradd --system --create-home ggc_user
sudo groupadd --system ggc_group
```

3. Verify that the user that runs the AWS IoT Greengrass Core software (typically `root`), has permission to run `sudo` with any user and any group.

   a. Run the following command to open the `/etc/sudoers` file.

   ```
   sudo visudo
   ```

   b. Verify that the permission for the user looks like the following example.

   ```
   root    ALL=(ALL:ALL) ALL
   ```

4. (Optional) To run containerized Lambda functions, you must enable cgroups v1, and you must enable and mount the *memory* and *devices* cgroups. If you don't plan to run containerized Lambda functions, you can skip this step.

   To enable these cgroups options, boot the device with the following Linux kernel parameters.

   ```
   cgroup_enable=memory cgroup_memory=1 systemd.unified_cgroup_hierarchy=0
   ```

   For information about viewing and setting kernel parameters for your device, see the documentation for your operating system and boot loader. Follow the instructions to permanently set the kernel parameters.
5. Install all other required dependencies on your device as indicated by the list of requirements in Device requirements.

Set up a Windows device

> **Note**
>
> This feature is available for v2.5.0 and later of the Greengrass nucleus component.

To set up a Windows device for AWS IoT Greengrass V2

1. Install the Java runtime, which AWS IoT Greengrass Core software requires to run. We recommend that you use Amazon Corretto or OpenJDK long-term support versions. Version 8 or higher is required.
2. Check whether Java is available on the PATH system variable, and add it if not. The LocalSystem account runs the AWS IoT Greengrass Core software, so you must add Java to the PATH system variable instead of the PATH user variable for your user. Do the following:

   a. Press the Windows key to open the start menu.
   b. Type **`environment variables`** to search for the system options from the start menu.
   c. In the start menu search results, choose **Edit the system environment variables** to open the **System properties** window.
   d. Choose **Environment variables...** to open the **Environment Variables** window.
   e. Under **System variables**, select **Path**, and then choose **Edit**. In the **Edit environment variable** window, you can view each path on a separate line.

f. Check if the path to the Java installation's `bin` folder is present. The path might look similar to the following example.

```
C:\\Program Files\\Amazon Corretto\\jdk11.0.13_8\\bin
```

g. If the Java installation's `bin` folder is missing from **Path**, choose **New** to add it, and then choose **OK**.

3. Open the Windows Command Prompt (`cmd.exe`) as an administrator.
4. Create the default user in the LocalSystem account on the Windows device. Replace *`password`* with a secure password.

```
net user /add ggc_user password
```

> **Tip**
>
> Depending on your Windows configuration, the user's password might be set to expire at a date in the future. To ensure your Greengrass applications continue to operate, track when the password expires, and update it before it expires. You can also set the user's password to never expire.
>
> - To check when a user and its password expire, run the following command.
>
>   ```
>   net user ggc_user | findstr /C:expires
>   ```
>
> - To set a user's password to never expire, run the following command.
>
>   ```
>   wmic UserAccount where "Name='ggc_user'" set PasswordExpires=False
>   ```
>
> - If you're using Windows 10 or later where the wmic command is deprecated, run the following PowerShell command.
>
>   ```
>   Get-CimInstance -Query "SELECT * from Win32_UserAccount WHERE name =
>    'ggc_user'" | Set-CimInstance -Property @{PasswordExpires="False"}
>   ```

5. Download and install the PsExec utility from Microsoft on the device.

6. Use the PsExec utility to store the user name and password for the default user in the Credential Manager instance for the LocalSystem account. Replace *password* with the user's password that you set earlier.

   ```
   psexec -s cmd /c cmdkey /generic:ggc_user /user:ggc_user /pass:password
   ```

   If the **PsExec License Agreement** opens, choose **Accept** to agree to the license and run the command.

   > **Note**
   >
   > On Windows devices, the LocalSystem account runs the Greengrass nucleus, and you must use the PsExec utility to store the default user information in the LocalSystem account. Using the Credential Manager application stores this information in the Windows account of the currently logged on user, instead of the LocalSystem account.

Download the AWS IoT Greengrass Core software

You can download the latest version of the AWS IoT Greengrass Core software from the following location:

- https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip

> **Note**
>
> You can download a specific version of the AWS IoT Greengrass Core software from the following location. Replace *version* with the version to download.
>
> ```
> https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-version.zip
> ```

To download the AWS IoT Greengrass Core software

1. On your core device, download the AWS IoT Greengrass Core software to a file named `greengrass-nucleus-latest.zip`.

Linux or Unix

```
curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-
latest.zip > greengrass-nucleus-latest.zip
```

Windows Command Prompt (CMD)

```
curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-
latest.zip > greengrass-nucleus-latest.zip
```

PowerShell

```
iwr -Uri https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-
latest.zip -OutFile greengrass-nucleus-latest.zip
```

By downloading this software, you agree to the Greengrass Core Software License Agreement.

2. (Optional) To verify the Greengrass nucleus software signature

> **Note**
>
> This feature is available with Greengrass nucleus version 2.9.5 and later.

a. Use the following command to verify your Greengrass nucleus artifact's signature:

Linux or Unix

```
jarsigner -verify -certs -verbose greengrass-nucleus-latest.zip
```

Windows Command Prompt (CMD)

The file name might look different depending on the JDK version you install. Replace *jdk17.0.6_10* with the JDK version you installed.

```
"C:\\Program Files\\Amazon Corretto\\jdk17.0.6_10\\bin\\jarsigner.exe" -
verify -certs -verbose greengrass-nucleus-latest.zip
```

PowerShell

The file name might look different depending on the JDK version you install. Replace *jdk17.0.6_10* with the JDK version you installed.

```
'C:\\Program Files\\Amazon Corretto\\jdk17.0.6_10\\bin\\jarsigner.exe' -
verify -certs -verbose greengrass-nucleus-latest.zip
```

b. The `jarsigner` invocation yields output that indicates the results of the verification.

   i. If the Greengrass nucleus zip file is signed, the output contains the following statement:

   ```
   jar verified.
   ```

   ii. If the Greengrass nucleus zip file isn't signed, the output contains the following statement:

   ```
   jar is unsigned.
   ```

c. If you provided the Jarsigner `-certs` option along with `-verify` and `-verbose` options, the output also includes detailed signer certificate information.

3. Unzip the AWS IoT Greengrass Core software to a folder on your device. Replace *GreengrassInstaller* with the folder that you want to use.

   Linux or Unix

   ```
   unzip greengrass-nucleus-latest.zip -d GreengrassInstaller && rm greengrass-
   nucleus-latest.zip
   ```

   Windows Command Prompt (CMD)

   ```
   mkdir GreengrassInstaller && tar -xf greengrass-nucleus-latest.zip -
   C GreengrassInstaller && del greengrass-nucleus-latest.zip
   ```

   PowerShell

   ```
   Expand-Archive -Path greengrass-nucleus-latest.zip -DestinationPath .\
   \GreengrassInstaller
   ```

```
rm greengrass-nucleus-latest.zip
```

4. (Optional) Run the following command to see the version of the AWS IoT Greengrass Core software.

```
java -jar ./GreengrassInstaller/lib/Greengrass.jar --version
```

Important

If you install a version of the Greengrass nucleus earlier than v2.4.0, don't remove this folder after you install the AWS IoT Greengrass Core software. The AWS IoT Greengrass Core software uses the files in this folder to run.
If you downloaded the latest version of the software, you install v2.4.0 or later, and you can remove this folder after you install the AWS IoT Greengrass Core software.

Download the AWS IoT fleet provisioning plugin

You can download the latest version of the AWS IoT fleet provisioning plugin from the following location:

- https://d2s8p88vqu9w66.cloudfront.net/releases/aws-greengrass-FleetProvisioningByClaim/fleetprovisioningbyclaim-latest.jar

Note

You can download a specific version of the AWS IoT fleet provisioning plugin from the following location. Replace *version* with the version to download. For more information about each version of the fleet provisioning plugin, see AWS IoT fleet provisioning plugin changelog.

```
https://d2s8p88vqu9w66.cloudfront.net/releases/aws-greengrass-
FleetProvisioningByClaim/fleetprovisioningbyclaim-version.jar
```

The fleet provisioning plugin is open source. To view its source code, see the AWS IoT fleet provisioning plugin on GitHub.

To download the AWS IoT fleet provisioning plugin

- On your device, download the AWS IoT fleet provisioning plugin to a file named `aws.greengrass.FleetProvisioningByClaim.jar`. Replace *`GreengrassInstaller`* with the folder that you want to use.

  Linux or Unix

  ```
  curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/aws-greengrass-
  FleetProvisioningByClaim/fleetprovisioningbyclaim-latest.jar
   > GreengrassInstaller/aws.greengrass.FleetProvisioningByClaim.jar
  ```

  Windows Command Prompt (CMD)

  ```
  curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/aws-greengrass-
  FleetProvisioningByClaim/fleetprovisioningbyclaim-latest.jar
   > GreengrassInstaller/aws.greengrass.FleetProvisioningByClaim.jar
  ```

  PowerShell

  ```
  iwr -Uri https://d2s8p88vqu9w66.cloudfront.net/releases/aws-greengrass-
  FleetProvisioningByClaim/fleetprovisioningbyclaim-latest.jar -
  OutFile GreengrassInstaller/aws.greengrass.FleetProvisioningByClaim.jar
  ```

  By downloading this software, you agree to the Greengrass Core Software License Agreement.

Install the AWS IoT Greengrass Core software

Run the installer with arguments that specify the following actions:

- Install from a partial configuration file that specifies to use the fleet provisioning plugin to provision AWS resources. The AWS IoT Greengrass Core software uses a configuration file that specifies the configuration of every Greengrass component on the device. The installer creates a complete configuration file from the partial configuration file that you provide and the AWS resources that the fleet provisioning plugin creates.
- Specify to use the `ggc_user` system user to run software components on the core device. On Linux devices, this command also specifies to use the `ggc_group` system group, and the installer creates the system user and group for you.

- Set up the AWS IoT Greengrass Core software as a system service that runs at boot. On Linux devices, this requires the Systemd init system.

> **Important**
>
> On Windows core devices, you must set up the AWS IoT Greengrass Core software as a system service.

For more information about the arguments that you can specify, see Installer arguments.

> **Note**
>
> If you are running AWS IoT Greengrass on a device with limited memory, you can control the amount of memory that AWS IoT Greengrass Core software uses. To control memory allocation, you can set JVM heap size options in the `jvmOptions` configuration parameter in your nucleus component. For more information, see Control memory allocation with JVM options.

To install the AWS IoT Greengrass Core software

1. Check the version of the AWS IoT Greengrass Core software.

   - Replace *`GreengrassInstaller`* with the path to the folder that contains the software.

   ```
   java -jar ./GreengrassInstaller/lib/Greengrass.jar --version
   ```

2. Use a text editor to create a configuration file named `config.yaml` to provide to the installer.

   For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

   ```
   nano GreengrassInstaller/config.yaml
   ```

   Copy the following YAML content into the file. This partial configuration file specifies parameters for the fleet provisioning plugin. For more information about the options that you can specify, see Configure the AWS IoT fleet provisioning plugin.

Linux or Unix

```
---
services:
  aws.greengrass.Nucleus:
    version: "2.18.4"
  aws.greengrass.FleetProvisioningByClaim:
    configuration:
      rootPath: "/greengrass/v2"
      awsRegion: "us-west-2"
      iotDataEndpoint: "device-data-prefix-ats.iot.us-west-2.amazonaws.com"
      iotCredentialEndpoint: "device-credentials-prefix.credentials.iot.us-
west-2.amazonaws.com"
      iotRoleAlias: "GreengrassCoreTokenExchangeRoleAlias"
      provisioningTemplate: "GreengrassFleetProvisioningTemplate"
      claimCertificatePath: "/greengrass/v2/claim-certs/claim.pem.crt"
      claimCertificatePrivateKeyPath: "/greengrass/v2/claim-certs/
claim.private.pem.key"
      rootCaPath: "/greengrass/v2/AmazonRootCA1.pem"
      templateParameters:
        ThingName: "MyGreengrassCore"
        ThingGroupName: "MyGreengrassCoreGroup"
```

Windows

```
---
services:
  aws.greengrass.Nucleus:
    version: "2.18.4"
  aws.greengrass.FleetProvisioningByClaim:
    configuration:
      rootPath: "C:\\greengrass\\v2"
      awsRegion: "us-west-2"
      iotDataEndpoint: "device-data-prefix-ats.iot.us-west-2.amazonaws.com"
      iotCredentialEndpoint: "device-credentials-prefix.credentials.iot.us-
west-2.amazonaws.com"
      iotRoleAlias: "GreengrassCoreTokenExchangeRoleAlias"
      provisioningTemplate: "GreengrassFleetProvisioningTemplate"
      claimCertificatePath: "C:\\greengrass\\v2\\claim-certs\\claim.pem.crt"
      claimCertificatePrivateKeyPath: "C:\\greengrass\\v2\\claim-certs\
\claim.private.pem.key"
      rootCaPath: "C:\\greengrass\\v2\\AmazonRootCA1.pem"
```

```
      templateParameters:
        ThingName: "MyGreengrassCore"
        ThingGroupName: "MyGreengrassCoreGroup"
```

Then, do the following:

- Replace *2.18.4* with the version of the AWS IoT Greengrass Core software.
- Replace each instance of */greengrass/v2* or *C:\greengrass\v2* with the Greengrass root folder.

  > **Note**
  >
  > On Windows devices, you must specify path separators as double backslashes (\\), such as `C:\\greengrass\\v2`.

- Replace *us-west-2* with the AWS Region where you created the provisioning template and other resources.
- Replace the `iotDataEndpoint` with your AWS IoT data endpoint.
- Replace the `iotCredentialEndpoint` with your AWS IoT credentials endpoint.
- Replace *GreengrassCoreTokenExchangeRoleAlias* with the name of the token exchange role alias.
- Replace *GreengrassFleetProvisioningTemplate* with the name of the fleet provisioning template.
- Replace the `claimCertificatePath` with the path to the claim certificate on the device.
- Replace the `claimCertificatePrivateKeyPath` with the path to the claim certificate private key on the device.
- Replace the template parameters (`templateParameters`) with the values to use to provision the device. This example refers to the example template that defines `ThingName` and `ThingGroupName` parameters.

> **Note**
>
> In this configuration file, you can customize other configuration options such as the ports and network proxy to use, as shown in the following example. For more information, see Greengrass nucleus configuration.

Linux or Unix

```
---
services:
  aws.greengrass.Nucleus:
    version: "2.18.4"
    configuration:
      mqtt:
        port: 443
      greengrassDataPlanePort: 443
      networkProxy:
        noProxyAddresses: "http://192.168.0.1,www.example.com"
        proxy:
          url: "http://my-proxy-server:1100"
          username: "Mary_Major"
          password: "pass@word1357"
  aws.greengrass.FleetProvisioningByClaim:
    configuration:
      rootPath: "/greengrass/v2"
      awsRegion: "us-west-2"
      iotDataEndpoint: "device-data-prefix-ats.iot.us-
west-2.amazonaws.com"
      iotCredentialEndpoint: "device-credentials-
prefix.credentials.iot.us-west-2.amazonaws.com"
      iotRoleAlias: "GreengrassCoreTokenExchangeRoleAlias"
      provisioningTemplate: "GreengrassFleetProvisioningTemplate"
      claimCertificatePath: "/greengrass/v2/claim-certs/claim.pem.crt"
      claimCertificatePrivateKeyPath: "/greengrass/v2/claim-certs/
claim.private.pem.key"
      rootCaPath: "/greengrass/v2/AmazonRootCA1.pem"
      templateParameters:
        ThingName: "MyGreengrassCore"
        ThingGroupName: "MyGreengrassCoreGroup"
      mqttPort: 443
      proxyUrl: "http://my-proxy-server:1100"
      proxyUserName: "Mary_Major"
      proxyPassword: "pass@word1357"
```

Windows

```
---
services:
```

```
  aws.greengrass.Nucleus:
    version: "2.18.4"
    configuration:
      mqtt:
        port: 443
      greengrassDataPlanePort: 443
      networkProxy:
        noProxyAddresses: "http://192.168.0.1,www.example.com"
        proxy:
          url: "http://my-proxy-server:1100"
          username: "Mary_Major"
          password: "pass@word1357"
  aws.greengrass.FleetProvisioningByClaim:
    configuration:
      rootPath: "C:\\greengrass\\v2"
      awsRegion: "us-west-2"
      iotDataEndpoint: "device-data-prefix-ats.iot.us-
west-2.amazonaws.com"
      iotCredentialEndpoint: "device-credentials-
prefix.credentials.iot.us-west-2.amazonaws.com"
      iotRoleAlias: "GreengrassCoreTokenExchangeRoleAlias"
      provisioningTemplate: "GreengrassFleetProvisioningTemplate"
      claimCertificatePath: "C:\\greengrass\\v2\\claim-certs\
\claim.pem.crt"
      claimCertificatePrivateKeyPath: "C:\\greengrass\\v2\\claim-certs\
\claim.private.pem.key"
      rootCaPath: "C:\\greengrass\\v2\\AmazonRootCA1.pem"
      templateParameters:
        ThingName: "MyGreengrassCore"
        ThingGroupName: "MyGreengrassCoreGroup"
      mqttPort: 443
      proxyUrl: "http://my-proxy-server:1100"
      proxyUserName: "Mary_Major"
      proxyPassword: "pass@word1357"
```

To use an HTTPS proxy, you must use version 1.1.0 or later of the fleet provisioning plugin. You must additionally specify the `rootCaPath` under `system`, as shown in the following example.

Linux or Unix

```
---
```

```
system:
  rootCaPath: "/greengrass/v2/AmazonRootCA1.pem"
services:
  ...
```

Windows

```
---
system:
  rootCaPath: "C:\\greengrass\\v2\\AmazonRootCA1.pem"
services:
  ...
```

3. Run the installer. Specify `--trusted-plugin` to provide the fleet provisioning plugin, and specify `--init-config` to provide the configuration file.

   - Replace */greengrass/v2* with the Greengrass root folder.
   - Replace each instance of *GreengrassInstaller* with the folder where you unpacked the installer.

   Linux or Unix

   ```
   sudo -E java -Droot="/greengrass/v2" -Dlog.store=FILE \
     -jar ./GreengrassInstaller/lib/Greengrass.jar \
     --trusted-plugin ./GreengrassInstaller/
   aws.greengrass.FleetProvisioningByClaim.jar \
     --init-config ./GreengrassInstaller/config.yaml \
     --component-default-user ggc_user:ggc_group \
     --setup-system-service true
   ```

   Windows Command Prompt (CMD)

   ```
   java -Droot="C:\greengrass\v2" "-Dlog.store=FILE" ^
     -jar ./GreengrassInstaller/lib/Greengrass.jar ^
     --trusted-plugin ./GreengrassInstaller/
   aws.greengrass.FleetProvisioningByClaim.jar ^
     --init-config ./GreengrassInstaller/config.yaml ^
     --component-default-user ggc_user ^
     --setup-system-service true
   ```

PowerShell

```
java -Droot="C:\greengrass\v2" "-Dlog.store=FILE" `
  -jar ./GreengrassInstaller/lib/Greengrass.jar `
  --trusted-plugin ./GreengrassInstaller/
aws.greengrass.FleetProvisioningByClaim.jar `
  --init-config ./GreengrassInstaller/config.yaml `
  --component-default-user ggc_user `
  --setup-system-service true
```

Important

On Windows core devices, you must specify `--setup-system-service true` to set up the AWS IoT Greengrass Core software as a system service.

If you specify `--setup-system-service true`, the installer prints `Successfully set up Nucleus as a system service` if it set up and ran the software as a system service. Otherwise, the installer doesn't output any message if it installs the software successfully.

Note

You can't use the `deploy-dev-tools` argument to deploy local development tools when you run the installer without the `--provision true` argument. For information about deploying the Greengrass CLI directly on your device, see Greengrass Command Line Interface.

4. Verify the installation by viewing the files in the root folder.

   Linux or Unix

   ```
   ls /greengrass/v2
   ```

   Windows Command Prompt (CMD)

   ```
   dir C:\greengrass\v2
   ```

PowerShell

```
ls C:\greengrass\v2
```

If the installation succeeded, the root folder contains several folders, such as `config`, `packages`, and `logs`.

If you installed the AWS IoT Greengrass Core software as a system service, the installer runs the software for you. Otherwise, you must run the software manually. For more information, see Run the AWS IoT Greengrass Core software.

For more information about how to configure and use the software and AWS IoT Greengrass, see the following:

- Configure the AWS IoT Greengrass Core software
- Develop AWS IoT Greengrass components
- Deploy AWS IoT Greengrass components to devices
- Greengrass Command Line Interface

Set up AWS IoT fleet provisioning for Greengrass core devices

To install the AWS IoT Greengrass Core software with fleet provisioning, you must first set up the following resources in your AWS account. These resources enable devices to register themselves with AWS IoT and operate as Greengrass core devices. Follow steps in this section once to create and configure these resources in your AWS account.

- A token exchange IAM role, which core devices use to authorize calls to AWS services.
- An AWS IoT role alias that points to the token exchange role.
- (Optional) An AWS IoT policy, which core devices use to authorize calls to the AWS IoT and AWS IoT Greengrass services. This AWS IoT policy must allow the `iot:AssumeRoleWithCertificate` permission for the AWS IoT role alias that points to the token exchange role.

  You can use a single AWS IoT policy for all core devices in your fleet, or you can configure your fleet provisioning template to create an AWS IoT policy for each core device.

- An AWS IoT fleet provisioning template. This template must specify the following:
  - An AWS IoT thing resource. You can specify a list of existing thing groups to deploy components to each device when it comes online.
  - An AWS IoT policy resource. This resource can define one of the following properties:
    - The name of an existing AWS IoT policy. If you choose this option, the core devices that you create from this template use the same AWS IoT policy, and you can manage their permissions as a fleet.
    - An AWS IoT policy document. If you choose this option, each core device that you create from this template uses a unique AWS IoT policy, and you can manage permissions for each individual core device.
  - An AWS IoT certificate resource. This certificate resource must use the `AWS::IoT::Certificate::Id` parameter to attach the certificate to the core device. For more information, see Just-in-time provisioning in the *AWS IoT Developer Guide*.
- An AWS IoT provisioning claim certificate and private key for the fleet provisioning template. You can embed this certificate and private key in devices during manufacturing, so the devices can register and provision themselves when they come online.

  Important

  Provisioning claim private keys should be secured at all times, including on Greengrass core devices. We recommend that you use Amazon CloudWatch metrics and logs to monitor for indications of misuse, such as unauthorized use of the claim certificate to provision devices. If you detect misuse, disable the provisioning claim certificate so that it can't be used for device provisioning. For more information, see Monitoring AWS IoT in the *AWS IoT Core Developer Guide*.

  To help you better manage the number of devices, and which devices, that register themselves in your AWS account, you can specify a pre-provisioning hook when you create a fleet provisioning template. A pre-provisioning hook is an AWS Lambda function that validates template parameters that devices provide during registration. For example, you might create a pre-provisioning hook that checks a device ID against a database to verify that the device has permission to provision. For more information, see Pre-provisioning hooks in the *AWS IoT Core Developer Guide*.
- An AWS IoT policy that you attach to the provisioning claim certificate to allow devices to register and use the fleet provisioning template.

Topics

- Create a token exchange role
- Create an AWS IoT policy
- Create a fleet provisioning template
- Create a provisioning claim certificate and private key

Create a token exchange role

Greengrass core devices use an IAM service role, called the *token exchange role*, to authorize calls to AWS services. The device uses the AWS IoT credentials provider to get temporary AWS credentials for this role, which allows the device to interact with AWS IoT, send logs to Amazon CloudWatch Logs, and download custom component artifacts from Amazon S3. For more information, see Authorize core devices to interact with AWS services.

You use an AWS IoT *role alias* to configure the token exchange role for Greengrass core devices. Role aliases enable you to change the token exchange role for a device but keep the device configuration the same. For more information, see Authorizing direct calls to AWS services in the *AWS IoT Core Developer Guide*.

In this section, you create a token exchange IAM role and an AWS IoT role alias that points to the role. If you have already set up a Greengrass core device, you can use its token exchange role and role alias instead of creating new ones.

To create a token exchange IAM role

1. Create an IAM role that your device can use as a token exchange role. Do the following:

   a. Create a file that contains the trust policy document that the token exchange role requires.

      For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

      ```
      nano device-role-trust-policy.json
      ```

      Copy the following JSON into the file.

      ```
      {
      ```

```
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "credentials.iot.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

b. Create the token exchange role with the trust policy document.

- Replace *GreengrassV2TokenExchangeRole* with the name of the IAM role to create.

```
aws iam create-role --role-name GreengrassV2TokenExchangeRole --assume-role-
policy-document file://device-role-trust-policy.json
```

The response looks similar to the following example, if the request succeeds.

```
{
  "Role": {
    "Path": "/",
    "RoleName": "GreengrassV2TokenExchangeRole",
    "RoleId": "AROAZ2YMUHYHK5OKM77FB",
    "Arn": "arn:aws:iam::123456789012:role/GreengrassV2TokenExchangeRole",
    "CreateDate": "2021-02-06T00:13:29+00:00",
    "AssumeRolePolicyDocument": {
      "Version": "2012-10-17",
      "Statement": [
        {
          "Effect": "Allow",
          "Principal": {
            "Service": "credentials.iot.amazonaws.com"
          },
          "Action": "sts:AssumeRole"
        }
      ]
    }
  }
```

c. Create a file that contains the access policy document that the token exchange role requires.

For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

```
nano device-role-access-policy.json
```

Copy the following JSON into the file.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:DescribeLogStreams",
        "s3:GetBucketLocation"
      ],
      "Resource": "*"
    }
  ]
}
```

> **Note**
>
> This access policy doesn't allow access to component artifacts in S3 buckets. To deploy custom components that define artifacts in Amazon S3, you must add permissions to the role to allow your core device to retrieve component artifacts. For more information, see Allow access to S3 buckets for component artifacts. If you don't yet have an S3 bucket for component artifacts, you can add these permissions later after you create a bucket.

d. Create the IAM policy from the policy document.

- Replace *GreengrassV2TokenExchangeRoleAccess* with the name of the IAM policy to create.

```
aws iam create-policy --policy-name GreengrassV2TokenExchangeRoleAccess --
policy-document file://device-role-access-policy.json
```

The response looks similar to the following example, if the request succeeds.

```
{
  "Policy": {
    "PolicyName": "GreengrassV2TokenExchangeRoleAccess",
    "PolicyId": "ANPAZ2YMUHYHACI7C5Z66",
    "Arn": "arn:aws:iam::123456789012:policy/
GreengrassV2TokenExchangeRoleAccess",
    "Path": "/",
    "DefaultVersionId": "v1",
    "AttachmentCount": 0,
    "PermissionsBoundaryUsageCount": 0,
    "IsAttachable": true,
    "CreateDate": "2021-02-06T00:37:17+00:00",
    "UpdateDate": "2021-02-06T00:37:17+00:00"
  }
}
```

e. Attach the IAM policy to the token exchange role.

- Replace *GreengrassV2TokenExchangeRole* with the name of the IAM role.
- Replace the policy ARN with the ARN of the IAM policy that you created in the previous step.

```
aws iam attach-role-policy --role-name GreengrassV2TokenExchangeRole --policy-
arn arn:aws:iam::123456789012:policy/GreengrassV2TokenExchangeRoleAccess
```

The command doesn't have any output if the request succeeds.

2. Create an AWS IoT role alias that points to the token exchange role.

- Replace *GreengrassCoreTokenExchangeRoleAlias* with the name of the role alias to create.
- Replace the role ARN with the ARN of the IAM role that you created in the previous step.

```
aws iot create-role-alias --role-alias GreengrassCoreTokenExchangeRoleAlias --role-
arn arn:aws:iam::123456789012:role/GreengrassV2TokenExchangeRole
```

The response looks similar to the following example, if the request succeeds.

```
{
  "roleAlias": "GreengrassCoreTokenExchangeRoleAlias",
  "roleAliasArn": "arn:aws:iot:us-west-2:123456789012:rolealias/
GreengrassCoreTokenExchangeRoleAlias"
}
```

> **Note**
>
> To create a role alias, you must have permission to pass the token exchange IAM role to AWS IoT. If you receive an error message when you try to create a role alias, check that your AWS user has this permission. For more information, see Granting a user permissions to pass a role to an AWS service in the *AWS Identity and Access Management User Guide*.

Create an AWS IoT policy

After you register a device as an AWS IoT thing, that device can use a digital certificate to authenticate with AWS. This certificate includes one or more AWS IoT policies that define the permissions that a device can use with the certificate. These policies allow the device to communicate with AWS IoT and AWS IoT Greengrass.

With AWS IoT fleet provisioning, devices connect to AWS IoT to create and download a device certificate. In the fleet provisioning template that you create in the next section, you can specify whether AWS IoT attaches the same AWS IoT policy to all devices' certificates, or creates a new policy for each device.

In this section, you create an AWS IoT policy that AWS IoT attaches to all devices' certificates. With this approach, you can manage permissions for all devices as a fleet. If you would rather create a new AWS IoT policy for each device, you can skip this section, and refer to the policy in it when you define your fleet template.

To create an AWS IoT policy

- Create an AWS IoT policy that defines the AWS IoT permissions for your fleet of Greengrass core devices. The following policy allows access to all MQTT topics and Greengrass operations, so your device works with custom applications and future changes that require new Greengrass operations. This policy also allows the `iot:AssumeRoleWithCertificate` permission, which allows your devices to use the token exchange role that you created in the previous section. You can restrict this policy down based on your use case. For more information, see Minimal AWS IoT policy for AWS IoT Greengrass V2 core devices.

  Do the following:

  a. Create a file that contains the AWS IoT policy document that Greengrass core devices require.

     For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

     ```
     nano greengrass-v2-iot-policy.json
     ```

     Copy the following JSON into the file.

     - Replace the `iot:AssumeRoleWithCertificate` resource with the ARN of the AWS IoT role alias that you created in the previous section.

     JSON

     ```
     {
       "Version":"2012-10-17",
       "Statement": [
         {
           "Effect": "Allow",
           "Action": [
             "iot:Publish",
             "iot:Subscribe",
             "iot:Receive",
             "iot:Connect",
             "greengrass:*"
           ],
     ```

```
      "Resource": [
        "*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": "iot:AssumeRoleWithCertificate",
      "Resource": "arn:aws:iot:us-east-1:123456789012:rolealias/
GreengrassCoreTokenExchangeRoleAlias"
    }
  ]
}
```

b. Create an AWS IoT policy from the policy document.

- Replace *GreengrassV2IoTThingPolicy* with the name of the policy to create.

```
aws iot create-policy --policy-name GreengrassV2IoTThingPolicy --policy-
document file://greengrass-v2-iot-policy.json
```

The response looks similar to the following example, if the request succeeds.

JSON

```
{
  "policyName": "GreengrassV2IoTThingPolicy",
  "policyArn": "arn:aws:iot:us-west-2:123456789012:policy/
GreengrassV2IoTThingPolicy",
  "policyDocument": "{
    \"Version\": \"2012-10-17\",
    \"Statement\": [
      {
        \"Effect\": \"Allow\",
        \"Action\": [
          \"iot:Publish\",
          \"iot:Subscribe\",
          \"iot:Receive\",
          \"iot:Connect\",
          \"greengrass:*\"
        ],
        \"Resource\": [
```

```
          \"*\"
        ]
      },
      {
        \"Effect\": \"Allow\",
        \"Action\": \"iot:AssumeRoleWithCertificate\",
        \"Resource\": \"arn:aws:iot:us-west-2:123456789012:rolealias/
GreengrassCoreTokenExchangeRoleAlias\"
      }
    ]
  }",
  "policyVersionId": "1"
}
```

Create a fleet provisioning template

AWS IoT fleet provisioning templates define how to provision AWS IoT things, policies, and certificates. To provision Greengrass core devices with the fleet provisioning plugin, you must create a template that specifies the following:

- An AWS IoT thing resource. You can specify a list of existing thing groups to deploy components to each device when it comes online.
- An AWS IoT policy resource. This resource can define one of the following properties:
  - The name of an existing AWS IoT policy. If you choose this option, the core devices that you create from this template use the same AWS IoT policy, and you can manage their permissions as a fleet.
  - An AWS IoT policy document. If you choose this option, each core device that you create from this template uses a unique AWS IoT policy, and you can manage permissions for each individual core device.
- An AWS IoT certificate resource. This certificate resource must use the `AWS::IoT::Certificate::Id` parameter to attach the certificate to the core device. For more information, see Just-in-time provisioning in the *AWS IoT Developer Guide*.

In the template, you can specify to add the AWS IoT thing to a list of existing thing groups. When the core device connects to AWS IoT Greengrass for the first time, it receives Greengrass deployments for each thing group where it's a member. You can use thing groups to deploy the

latest software to each device as soon as it comes online. For more information, see Deploy AWS IoT Greengrass components to devices.

The AWS IoT service requires permissions to create and update AWS IoT resources in your AWS account when provisioning devices. To give the AWS IoT service access, you create an IAM role and provide it when you create the template. AWS IoT provides an managed policy, AWSIoTThingsRegistration, that allows access to all permissions that AWS IoT might use when provisioning devices. You can use this managed policy, or create a custom policy that scopes down the permissions in the managed policy for your use case.

In this section, you create an IAM role that allows AWS IoT to provision resources for devices, and you create a fleet provisioning template that uses that IAM role.

To create a fleet provisioning template

1. Create an IAM role that AWS IoT can assume to provision resources in your AWS account. Do the following:

   a. Create a file that contains the trust policy document that allows AWS IoT to assume the role.

      For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

      ```
      nano aws-iot-trust-policy.json
      ```

      Copy the following JSON into the file.

      JSON

      ```
      {
        "Version":"2012-10-17",
        "Statement": [
          {
            "Effect": "Allow",
            "Principal": {
              "Service": "iot.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
          }
      ```

```
  ]
}
```

b. Create an IAM role with the trust policy document.

- Replace *GreengrassFleetProvisioningRole* with the name of the IAM role to create.

```
aws iam create-role --role-name GreengrassFleetProvisioningRole --assume-role-
policy-document file://aws-iot-trust-policy.json
```

The response looks similar to the following example, if the request succeeds.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:Connect",
        "iot:Publish",
        "iot:Subscribe",
        "iot:Receive"
      ],
      "Resource": "*"
    }
  ]
}
```

c. Review the AWSIoTThingsRegistration policy, which allows access to all permissions that AWS IoT might use when provisioning devices. You can use this managed policy, or create a custom policy that defines scoped-down permissions for your use case. If you choose to create a custom policy, do so now.

d. Attach the IAM policy to the fleet provisioning role.

- Replace *GreengrassFleetProvisioningRole* with the name of the IAM role.

- If you created a custom policy in the previous step, replace the policy ARN with the ARN of the IAM policy to use.

```
aws iam attach-role-policy --role-name GreengrassFleetProvisioningRole --
policy-arn arn:aws:iam::aws:policy/service-role/AWSIoTThingsRegistration
```

The command doesn't have any output if the request succeeds.

2. (Optional) Create a *pre-provisioning hook*, which is an AWS Lambda function that validates template parameters that devices provide during registration. You can use a pre-provisioning hook to gain more control over which and how many devices onboard in your AWS account. For more information, see Pre-provisioning hooks in the *AWS IoT Core Developer Guide*.
3. Create a fleet provisioning template. Do the following:

   a. Create a file to contain the provisioning template document.

   For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

   ```
   nano greengrass-fleet-provisioning-template.json
   ```

   Write the provisioning template document. You can start from the following example provisioning template, which specifies to create an AWS IoT thing with the following properties:

   - The thing's name is the value that you specify in the `ThingName` template parameter.
   - The thing is a member of the thing group that you specify in the `ThingGroupName` template parameter. The thing group must exist in your AWS account.
   - The thing's certificate has the AWS IoT policy named `GreengrassV2IoTThingPolicy` attached to it.

   For more information, see Provisioning templates in the *AWS IoT Core Developer Guide*.

   ```
   {
     "Parameters": {
       "ThingName": {
         "Type": "String"
   ```

```
    },
    "ThingGroupName": {
      "Type": "String"
    },
    "AWS::IoT::Certificate::Id": {
      "Type": "String"
    }
  },
  "Resources": {
    "MyThing": {
      "OverrideSettings": {
        "AttributePayload": "REPLACE",
        "ThingGroups": "REPLACE",
        "ThingTypeName": "REPLACE"
      },
      "Properties": {
        "AttributePayload": {},
        "ThingGroups": [
          {
            "Ref": "ThingGroupName"
          }
        ],
        "ThingName": {
          "Ref": "ThingName"
        }
      },
      "Type": "AWS::IoT::Thing"
    },
    "MyPolicy": {
      "Properties": {
        "PolicyName": "GreengrassV2IoTThingPolicy"
      },
      "Type": "AWS::IoT::Policy"
    },
    "MyCertificate": {
      "Properties": {
        "CertificateId": {
          "Ref": "AWS::IoT::Certificate::Id"
        },
        "Status": "Active"
      },
      "Type": "AWS::IoT::Certificate"
    }
  }
```

```
}
```

> **Note**
>
> *MyThing*, *MyPolicy*, and *MyCertificate* are arbitrary names that identify each resource specification in the fleet provisioning template. AWS IoT doesn't use these names in the resources that it creates from the template. You can use these names or replace them with values that help you identify each resource in the template.

b. Create the fleet provisioning template from the provisioning template document.

- Replace *GreengrassFleetProvisioningTemplate* with the name of the template to create.
- Replace the template description with a description for your template.
- Replace the provisioning role ARN with the ARN of the role that you created earlier.

Linux or Unix

```
aws iot create-provisioning-template \
  --template-name GreengrassFleetProvisioningTemplate \
  --description "A provisioning template for Greengrass core devices." \
  --provisioning-role-arn "arn:aws:iam::123456789012:role/
GreengrassFleetProvisioningRole" \
  --template-body file://greengrass-fleet-provisioning-template.json \
  --enabled
```

Windows Command Prompt (CMD)

```
aws iot create-provisioning-template ^
  --template-name GreengrassFleetProvisioningTemplate ^
  --description "A provisioning template for Greengrass core devices." ^
  --provisioning-role-arn "arn:aws:iam::123456789012:role/
GreengrassFleetProvisioningRole" ^
  --template-body file://greengrass-fleet-provisioning-template.json ^
  --enabled
```

PowerShell

```
aws iot create-provisioning-template `
  --template-name GreengrassFleetProvisioningTemplate `
  --description "A provisioning template for Greengrass core devices." `
  --provisioning-role-arn "arn:aws:iam::123456789012:role/
GreengrassFleetProvisioningRole" `
  --template-body file://greengrass-fleet-provisioning-template.json `
  --enabled
```

Note

If you created a pre-provisioning hook, specify the ARN of the pre-provisioning hook's Lambda function with the `--pre-provisioning-hook` argument.

```
--pre-provisioning-hook targetArn=arn:aws:lambda:us-
west-2:123456789012:function:GreengrassPreProvisioningHook
```

The response looks similar to the following example, if the request succeeds.

```
{
    "templateArn": "arn:aws:iot:us-west-2:123456789012:provisioningtemplate/
GreengrassFleetProvisioningTemplate",
    "templateName": "GreengrassFleetProvisioningTemplate",
    "defaultVersionId": 1
}
```

Create a provisioning claim certificate and private key

Claim certificates are X.509 certificates that allow devices to register as AWS IoT things and retrieve a unique X.509 device certificate to use for regular operations. After you create a claim certificate, you attach an AWS IoT policy that allows devices to use it to create unique device certificates and provision with a fleet provisioning template. Devices with the claim certificate can provision using only the provisioning template that you allow in the AWS IoT policy.

In this section, you create the claim certificate and configure it for devices to use with the fleet provisioning template that you created in the previous section.

Important

Provisioning claim private keys should be secured at all times, including on Greengrass core devices. We recommend that you use Amazon CloudWatch metrics and logs to monitor for indications of misuse, such as unauthorized use of the claim certificate to provision devices. If you detect misuse, disable the provisioning claim certificate so that it can't be used for device provisioning. For more information, see Monitoring AWS IoT in the *AWS IoT Core Developer Guide*.

To help you better manage the number of devices, and which devices, that register themselves in your AWS account, you can specify a pre-provisioning hook when you create a fleet provisioning template. A pre-provisioning hook is an AWS Lambda function that validates template parameters that devices provide during registration. For example, you might create a pre-provisioning hook that checks a device ID against a database to verify that the device has permission to provision. For more information, see Pre-provisioning hooks in the *AWS IoT Core Developer Guide*.

To create a provisioning claim certificate and private key

1. Create a folder where you download the claim certificate and private key.

   ```
   mkdir claim-certs
   ```

2. Create and save a certificate and private key to use for provisioning. AWS IoT provides client certificates that are signed by the Amazon Root certificate authority (CA).

   Linux or Unix

   ```
   aws iot create-keys-and-certificate \
     --certificate-pem-outfile "claim-certs/claim.pem.crt" \
     --public-key-outfile "claim-certs/claim.public.pem.key" \
     --private-key-outfile "claim-certs/claim.private.pem.key" \
     --set-as-active
   ```

Windows Command Prompt (CMD)

```
aws iot create-keys-and-certificate ^
  --certificate-pem-outfile "claim-certs/claim.pem.crt" ^
  --public-key-outfile "claim-certs/claim.public.pem.key" ^
  --private-key-outfile "claim-certs/claim.private.pem.key" ^
  --set-as-active
```

PowerShell

```
aws iot create-keys-and-certificate `
  --certificate-pem-outfile "claim-certs/claim.pem.crt" `
  --public-key-outfile "claim-certs/claim.public.pem.key" `
  --private-key-outfile "claim-certs/claim.private.pem.key" `
  --set-as-active
```

The response contains information about the certificate, if the request succeeds. Save the certificate's ARN to use later.

3. Create and attach an AWS IoT policy that allows devices to use the certificate to create unique device certificates and provision with the fleet provisioning template. The following policy allows access to the device provisioning MQTT API. For more information, see Device provisioning MQTT API in the *AWS IoT Core Developer Guide*.

   Do the following:

   a. Create a file that contains the AWS IoT policy document that Greengrass core devices require.

      For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

      ```
      nano greengrass-provisioning-claim-iot-policy.json
      ```

      Copy the following JSON into the file.

      - Replace each instance of *`region`* with the AWS Region where you set up fleet provisioning.
      - Replace each instance of *`account-id`* with your AWS account ID.

- Replace each instance of *GreengrassFleetProvisioningTemplate* with the name of the fleet provisioning template that you created in the previous section.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "iot:Connect",
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "iot:Publish",
        "iot:Receive"
      ],
      "Resource": [
      "arn:aws:iot:us-east-1:123456789012:topic/$aws/certificates/create/
*",
  "arn:aws:iot:us-east-1:123456789012:topic/$aws/provisioning-
templates/GreengrassFleetProvisioningTemplate/provision/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": "iot:Subscribe",
      "Resource": [
      "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/certificates/
create/*",
  "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/provisioning-
templates/GreengrassFleetProvisioningTemplate/provision/*"
      ]
    }
  ]
}
```

b. Create an AWS IoT policy from the policy document.

- Replace *GreengrassProvisioningClaimPolicy* with the name of the policy to create.

```
aws iot create-policy --policy-name GreengrassProvisioningClaimPolicy --policy-
document file://greengrass-provisioning-claim-iot-policy.json
```

The response looks similar to the following example, if the request succeeds.

JSON

```
{
  "policyName": "GreengrassProvisioningClaimPolicy",
  "policyArn": "arn:aws:iot:us-west-2:123456789012:policy/
GreengrassProvisioningClaimPolicy",
  "policyDocument": "{
    \"Version\": \"2012-10-17\",
    \"Statement\": [
      {
        \"Effect\": \"Allow\",
        \"Action\": \"iot:Connect\",
        \"Resource\": \"*\"
      },
      {
        \"Effect\": \"Allow\",
        \"Action\": [
          \"iot:Publish\",
          \"iot:Receive\"
        ],
        \"Resource\": [
        \"arn:aws:iot:us-east-1:123456789012:topic/$aws/certificates/
create/*\",
  \"arn:aws:iot:us-east-1:123456789012:topic/$aws/provisioning-
templates/GreengrassFleetProvisioningTemplate/provision/*\"
        ]
      },
      {
        \"Effect\": \"Allow\",
        \"Action\": \"iot:Subscribe\",
        \"Resource\": [
```

```
        \"arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/
certificates/create/*\",
  \"arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/provisioning-
templates/GreengrassFleetProvisioningTemplate/provision/*\"
        ]
      }
    ]
  }",
  "policyVersionId": "1"
}
```

4. Attach the AWS IoT policy to the provisioning claim certificate.

   - Replace *GreengrassProvisioningClaimPolicy* with the name of the policy to attach.
   - Replace the target ARN with the ARN of the provisioning claim certificate.

   ```
   aws iot attach-policy --policy-name GreengrassProvisioningClaimPolicy --
   target arn:aws:iot:us-west-2:123456789012:cert/
   aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4
   ```

   The command doesn't have any output if the request succeeds.

You now have a provisioning claim certificate and private key that devices can use to register with AWS IoT and provision themselves as Greengrass core devices. You can embed the claim certificate and private key in devices during manufacturing, or copy the certificate and key to devices before you install the AWS IoT Greengrass Core software. For more information, see Install AWS IoT Greengrass Core software with AWS IoT fleet provisioning.

Configure the AWS IoT fleet provisioning plugin

The AWS IoT fleet provisioning plugin provides the following configuration parameters that you can customize when you install the AWS IoT Greengrass Core software with fleet provisioning.

`rootPath`

The path to the folder to use as the root for the AWS IoT Greengrass Core software.

`awsRegion`

The AWS Region that the fleet provisioning plugin uses to provision AWS resources.

`iotDataEndpoint`

The AWS IoT data endpoint for your AWS account.

`iotCredentialEndpoint`

The AWS IoT credentials endpoint for your AWS account.

`iotRoleAlias`

The AWS IoT role alias that points to a token exchange IAM role. The AWS IoT credentials provider assumes this role to allow the Greengrass core device to interact with AWS services. For more information, see Authorize core devices to interact with AWS services.

`provisioningTemplate`

The AWS IoT fleet provisioning template to use to provision AWS resources. This template must specify the following:

- An AWS IoT thing resource. You can specify a list of existing thing groups to deploy components to each device when it comes online.
- An AWS IoT policy resource. This resource can define one of the following properties:
  - The name of an existing AWS IoT policy. If you choose this option, the core devices that you create from this template use the same AWS IoT policy, and you can manage their permissions as a fleet.
  - An AWS IoT policy document. If you choose this option, each core device that you create from this template uses a unique AWS IoT policy, and you can manage permissions for each individual core device.
- An AWS IoT certificate resource. This certificate resource must use the `AWS::IoT::Certificate::Id` parameter to attach the certificate to the core device. For more information, see Just-in-time provisioning in the *AWS IoT Developer Guide*.

For more information, see Provisioning templates in the *AWS IoT Core Developer Guide*.

`claimCertificatePath`

The path to the provisioning claim certificate for the provisioning template that you specify in `provisioningTemplate`. For more information, see CreateProvisioningClaim in the *AWS IoT Core API Reference*.

`claimCertificatePrivateKeyPath`

The path to the provisioning claim certificate private key for the provisioning template that you specify in `provisioningTemplate`. For more information, see CreateProvisioningClaim in the *AWS IoT Core API Reference*.

> **Important**
>
> Provisioning claim private keys should be secured at all times, including on Greengrass core devices. We recommend that you use Amazon CloudWatch metrics and logs to monitor for indications of misuse, such as unauthorized use of the claim certificate to provision devices. If you detect misuse, disable the provisioning claim certificate so that it can't be used for device provisioning. For more information, see Monitoring AWS IoT in the *AWS IoT Core Developer Guide*.
>
> To help you better manage the number of devices, and which devices, that register themselves in your AWS account, you can specify a pre-provisioning hook when you create a fleet provisioning template. A pre-provisioning hook is an AWS Lambda function that validates template parameters that devices provide during registration. For example, you might create a pre-provisioning hook that checks a device ID against a database to verify that the device has permission to provision. For more information, see Pre-provisioning hooks in the *AWS IoT Core Developer Guide*.

`rootCaPath`

The path to the Amazon root certificate authority (CA) certificate.

`templateParameters`

(Optional) The map of parameters to provide to the fleet provisioning template. For more information, see Provisioning templates' parameters section in the *AWS IoT Core Developer Guide*.

`deviceId`

(Optional) The device identifier to use as the client ID when the fleet provisioning plugin creates an MQTT connection to AWS IoT.

Default: A random UUID.

`mqttPort`

(Optional) The port to use for MQTT connections.

Default: `8883`

`proxyUrl`

(Optional) The URL of the proxy server in the format `scheme://userinfo@host:port`. To use an HTTPS proxy, you must use version 1.1.0 or later of the fleet provisioning plugin.

- `scheme` – The scheme, which must be `http` or `https`.

  Important

  Greengrass core devices must run Greengrass nucleus v2.5.0 or later to use HTTPS proxies.
  If you configure an HTTPS proxy, you must add the proxy server CA certificate to the core device's Amazon root CA certificate. For more information, see Enable the core device to trust an HTTPS proxy.

- `userinfo` – (Optional) The user name and password information. If you specify this information in the `url`, the Greengrass core device ignores the `username` and `password` fields.
- `host` – The host name or IP address of the proxy server.
- `port` – (Optional) The port number. If you don't specify the port, then the Greengrass core device uses the following default values:
  - `http` – 80
  - `https` – 443

`proxyUserName`

(Optional) The user name that authenticates the proxy server.

`proxyPassword`

(Optional) The user name that authenticates the proxy server.

csrPath

(Optional) The path to the certificate signing request (CSR) file to use to create the device certificate from a CSR. For more information, see Provisioning by claim in the *AWS IoT Core developer guide*.

csrPrivateKeyPath

(Optional, required if `csrPath` is declared) The path to the private key used to generate the CSR. The private key must have been used to generate the CSR. For more information, see Provisioning by claim in the *AWS IoT Core developer guide*.

certificatePath

(Optional) The path to use to save the downloaded device certificate.

privateKeyPath

(Optional) The path to use to save the downloaded device private key.

AWS IoT fleet provisioning plugin changelog

The following table describes the changes in each version of the AWS IoT fleet provisioning by claim plugin (`aws.greengrass.FleetProvisioningByClaim`).

Version	Changes
1.2.2	Bug fixes and improvements • Adds support for custom core device certificate (`certificatePath`) and private key (`privateKeyPath`) paths.
1.2.1	Bug fixes and improvements • Fixes an issue where the fleet provisioning plugin is offline during a Greengrass nucleus startup. The fleet provisioning plugin now indefinit ely retries MQTT connect calls.
1.2.0	Bug fixes and improvements • Adds support for device provisioning via certificate signing request with configurable private key path. • Minor fixes and improvements.
1.1.0	Bug fixes and improvements • Adds support for additional file path formats when you configure the plugin on Windows devices.

Version	Changes
	• Adds support for HTTPS network proxy configurations. For more information, see Connect on port 443 or through a network proxy and Enable the core device to trust an HTTPS proxy.
1.0.0	Initial version.

Install AWS IoT Greengrass Core software with custom resource provisioning

This feature is available for v2.4.0 and later of the Greengrass nucleus component.

The AWS IoT Greengrass Core software installer provides a Java interface that you can implement in a custom plugin that provisions required AWS resources. You can develop a provisioning plugin to use custom X.509 client certificates or to run complex provisioning steps that other installation processes don't support. For more information, see Create your own client certificates in the *AWS IoT Core Developer Guide*.

To run a custom provisioning plugin when you install the AWS IoT Greengrass Core software, you create a JAR file that you provide to the installer. The installer runs the plugin, and the plugin returns a provisioning configuration that defines the AWS resources for the Greengrass core device. The installer uses this information to configure the AWS IoT Greengrass Core software on the device. For more information, see Develop custom provisioning plugins.

> **Important**
>
> Before you download the AWS IoT Greengrass Core software, check that your core device meets the requirements to install and run the AWS IoT Greengrass Core software v2.0.

Topics

- Prerequisites
- Set up the device environment
- Download the AWS IoT Greengrass Core software
- Install the AWS IoT Greengrass Core software

- Develop custom provisioning plugins

Prerequisites

To install the AWS IoT Greengrass Core software with custom provisioning, you must have the following:

- A JAR file for a custom provisioning plugin that implements the `DeviceIdentityInterface`. The custom provisioning plugin must return values for each system and nucleus configuration parameter. Otherwise, you must provide those values in the configuration file during installation. For more information, see Develop custom provisioning plugins.

Set up the device environment

Follow the steps in this section to set up a Linux or Windows device to use as your AWS IoT Greengrass core device.

Set up a Linux device

To set up a Linux device for AWS IoT Greengrass V2

1. Install the Java runtime, which AWS IoT Greengrass Core software requires to run. We recommend that you use Amazon Corretto or OpenJDK long-term support versions. Version 8 or higher is required. The following commands show you how to install OpenJDK on your device.

   - For Debian-based or Ubuntu-based distributions:

     ```
     sudo apt install default-jdk
     ```

   - For Red Hat-based distributions:

     ```
     sudo yum install java-11-openjdk-devel
     ```

   - For Amazon Linux 2:

     ```
     sudo amazon-linux-extras install java-openjdk11
     ```

   - For Amazon Linux 2023:

```
sudo dnf install java-11-amazon-corretto -y
```

When the installation completes, run the following command to verify that Java runs on your Linux device.

```
java -version
```

The command prints the version of Java that runs on the device. For example, on a Debian-based distribution, the output might look similar to the following sample.

```
openjdk version "11.0.9.1" 2020-11-04
OpenJDK Runtime Environment (build 11.0.9.1+1-post-Debian-1deb10u2)
OpenJDK 64-Bit Server VM (build 11.0.9.1+1-post-Debian-1deb10u2, mixed mode)
```

2. (Optional) Create the default system user and group that runs components on the device. You can also choose to let the AWS IoT Greengrass Core software installer create this user and group during installation with the `--component-default-user` installer argument. For more information, see Installer arguments.

```
sudo useradd --system --create-home ggc_user
sudo groupadd --system ggc_group
```

3. Verify that the user that runs the AWS IoT Greengrass Core software (typically `root`), has permission to run `sudo` with any user and any group.

   a. Run the following command to open the `/etc/sudoers` file.

   ```
   sudo visudo
   ```

   b. Verify that the permission for the user looks like the following example.

   ```
   root    ALL=(ALL:ALL) ALL
   ```

4. (Optional) To run containerized Lambda functions, you must enable cgroups v1, and you must enable and mount the *memory* and *devices* cgroups. If you don't plan to run containerized Lambda functions, you can skip this step.

   To enable these cgroups options, boot the device with the following Linux kernel parameters.

```
cgroup_enable=memory cgroup_memory=1 systemd.unified_cgroup_hierarchy=0
```

For information about viewing and setting kernel parameters for your device, see the documentation for your operating system and boot loader. Follow the instructions to permanently set the kernel parameters.

5. Install all other required dependencies on your device as indicated by the list of requirements in Device requirements.

Set up a Windows device

> **Note**
>
> This feature is available for v2.5.0 and later of the Greengrass nucleus component.

To set up a Windows device for AWS IoT Greengrass V2

1. Install the Java runtime, which AWS IoT Greengrass Core software requires to run. We recommend that you use Amazon Corretto or OpenJDK long-term support versions. Version 8 or higher is required.
2. Check whether Java is available on the PATH system variable, and add it if not. The LocalSystem account runs the AWS IoT Greengrass Core software, so you must add Java to the PATH system variable instead of the PATH user variable for your user. Do the following:

   a. Press the Windows key to open the start menu.

   b. Type **`environment variables`** to search for the system options from the start menu.

   c. In the start menu search results, choose **Edit the system environment variables** to open the **System properties** window.

   d. Choose **Environment variables...** to open the **Environment Variables** window.

   e. Under **System variables**, select **Path**, and then choose **Edit**. In the **Edit environment variable** window, you can view each path on a separate line.

   f. Check if the path to the Java installation's `bin` folder is present. The path might look similar to the following example.

   ```
   C:\\Program Files\\Amazon Corretto\\jdk11.0.13_8\\bin
   ```

g. If the Java installation's `bin` folder is missing from **Path**, choose **New** to add it, and then choose **OK**.

3. Open the Windows Command Prompt (`cmd.exe`) as an administrator.
4. Create the default user in the LocalSystem account on the Windows device. Replace *password* with a secure password.

```
net user /add ggc_user password
```

> **Tip**
>
> Depending on your Windows configuration, the user's password might be set to expire at a date in the future. To ensure your Greengrass applications continue to operate, track when the password expires, and update it before it expires. You can also set the user's password to never expire.
>
> - To check when a user and its password expire, run the following command.
>
>   ```
>   net user ggc_user | findstr /C:expires
>   ```
>
> - To set a user's password to never expire, run the following command.
>
>   ```
>   wmic UserAccount where "Name='ggc_user'" set PasswordExpires=False
>   ```
>
> - If you're using Windows 10 or later where the wmic command is deprecated, run the following PowerShell command.
>
>   ```
>   Get-CimInstance -Query "SELECT * from Win32_UserAccount WHERE name =
>    'ggc_user'" | Set-CimInstance -Property @{PasswordExpires="False"}
>   ```

5. Download and install the PsExec utility from Microsoft on the device.
6. Use the PsExec utility to store the user name and password for the default user in the Credential Manager instance for the LocalSystem account. Replace *password* with the user's password that you set earlier.

```
psexec -s cmd /c cmdkey /generic:ggc_user /user:ggc_user /pass:password
```

If the **PsExec License Agreement** opens, choose **Accept** to agree to the license and run the command.

> **Note**
>
> On Windows devices, the LocalSystem account runs the Greengrass nucleus, and you must use the PsExec utility to store the default user information in the LocalSystem account. Using the Credential Manager application stores this information in the Windows account of the currently logged on user, instead of the LocalSystem account.

Download the AWS IoT Greengrass Core software

You can download the latest version of the AWS IoT Greengrass Core software from the following location:

- https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip

> **Note**
>
> You can download a specific version of the AWS IoT Greengrass Core software from the following location. Replace *version* with the version to download.
>
> ```
> https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-version.zip
> ```

To download the AWS IoT Greengrass Core software

1. On your core device, download the AWS IoT Greengrass Core software to a file named `greengrass-nucleus-latest.zip`.

   Linux or Unix

   ```
   curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-
   latest.zip > greengrass-nucleus-latest.zip
   ```

Windows Command Prompt (CMD)

```
curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-
latest.zip > greengrass-nucleus-latest.zip
```

PowerShell

```
iwr -Uri https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-
latest.zip -OutFile greengrass-nucleus-latest.zip
```

By downloading this software, you agree to the Greengrass Core Software License Agreement.

2. (Optional) To verify the Greengrass nucleus software signature

> **Note**
>
> This feature is available with Greengrass nucleus version 2.9.5 and later.

a. Use the following command to verify your Greengrass nucleus artifact's signature:

Linux or Unix

```
jarsigner -verify -certs -verbose greengrass-nucleus-latest.zip
```

Windows Command Prompt (CMD)

The file name might look different depending on the JDK version you install. Replace *jdk17.0.6_10* with the JDK version you installed.

```
"C:\\Program Files\\Amazon Corretto\\jdk17.0.6_10\\bin\\jarsigner.exe" -
verify -certs -verbose greengrass-nucleus-latest.zip
```

PowerShell

The file name might look different depending on the JDK version you install. Replace *jdk17.0.6_10* with the JDK version you installed.

```
'C:\\Program Files\\Amazon Corretto\\jdk17.0.6_10\\bin\\jarsigner.exe' -
verify -certs -verbose greengrass-nucleus-latest.zip
```

b. The `jarsigner` invocation yields output that indicates the results of the verification.

   i. If the Greengrass nucleus zip file is signed, the output contains the following statement:

   ```
   jar verified.
   ```

   ii. If the Greengrass nucleus zip file isn't signed, the output contains the following statement:

   ```
   jar is unsigned.
   ```

c. If you provided the Jarsigner `-certs` option along with `-verify` and `-verbose` options, the output also includes detailed signer certificate information.

3. Unzip the AWS IoT Greengrass Core software to a folder on your device. Replace *GreengrassInstaller* with the folder that you want to use.

   Linux or Unix

   ```
   unzip greengrass-nucleus-latest.zip -d GreengrassInstaller && rm greengrass-
   nucleus-latest.zip
   ```

   Windows Command Prompt (CMD)

   ```
   mkdir GreengrassInstaller && tar -xf greengrass-nucleus-latest.zip -
   C GreengrassInstaller && del greengrass-nucleus-latest.zip
   ```

   PowerShell

   ```
   Expand-Archive -Path greengrass-nucleus-latest.zip -DestinationPath .\
   \GreengrassInstaller
   rm greengrass-nucleus-latest.zip
   ```

4. (Optional) Run the following command to see the version of the AWS IoT Greengrass Core software.

```
java -jar ./GreengrassInstaller/lib/Greengrass.jar --version
```

> **Important**
>
> If you install a version of the Greengrass nucleus earlier than v2.4.0, don't remove this folder after you install the AWS IoT Greengrass Core software. The AWS IoT Greengrass Core software uses the files in this folder to run.
> If you downloaded the latest version of the software, you install v2.4.0 or later, and you can remove this folder after you install the AWS IoT Greengrass Core software.

Install the AWS IoT Greengrass Core software

Run the installer with arguments that specify the following actions:

- Install from a partial configuration file that specifies to use your custom provisioning plugin to provision AWS resources. The AWS IoT Greengrass Core software uses a configuration file that specifies the configuration of every Greengrass component on the device. The installer creates a complete configuration file from the partial configuration file that you provide and the AWS resources that the custom provisioning plugin creates.
- Specify to use the `ggc_user` system user to run software components on the core device. On Linux devices, this command also specifies to use the `ggc_group` system group, and the installer creates the system user and group for you.
- Set up the AWS IoT Greengrass Core software as a system service that runs at boot. On Linux devices, this requires the Systemd init system.

> **Important**
>
> On Windows core devices, you must set up the AWS IoT Greengrass Core software as a system service.

For more information about the arguments that you can specify, see Installer arguments.

> **Note**
>
> If you are running AWS IoT Greengrass on a device with limited memory, you can control the amount of memory that AWS IoT Greengrass Core software uses. To control memory allocation, you can set JVM heap size options in the `jvmOptions` configuration parameter in your nucleus component. For more information, see Control memory allocation with JVM options.

To install the AWS IoT Greengrass Core software (Linux)

1. Check the version of the AWS IoT Greengrass Core software.

   - Replace *GreengrassInstaller* with the path to the folder that contains the software.

   ```
   java -jar ./GreengrassInstaller/lib/Greengrass.jar --version
   ```

2. Use a text editor to create a configuration file named `config.yaml` to provide to the installer.

   For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

   ```
   nano GreengrassInstaller/config.yaml
   ```

   Copy the following YAML content into the file.

   ```
   ---
   system:
     rootpath: "/greengrass/v2"
     # The following values are optional. Return them from the provisioning plugin or
    set them here.
     # certificateFilePath: ""
     # privateKeyPath: ""
     # rootCaPath: ""
     # thingName: ""
   services:
     aws.greengrass.Nucleus:
       version: "2.18.4"
       configuration:
   ```

```
      # The following values are optional. Return them from the provisioning plugin
or set them here.
      # awsRegion: ""
      # iotRoleAlias: ""
      # iotDataEndpoint: ""
      # iotCredEndpoint: ""
  com.example.CustomProvisioning:
    configuration:
      # You can specify configuration parameters to provide to your plugin.
      # pluginParameter: ""
```

Then, do the following:

- Replace *2.18.4* with the version of the AWS IoT Greengrass Core software.
- Replace each instance of */greengrass/v2* with the Greengrass root folder.
- (Optional) Specify system and nucleus configuration values. You must set these values if your provisioning plugin doesn't provide them.
- (Optional) Specify configuration parameters to provide to your provisioning plugin.

> **Note**
>
> In this configuration file, you can customize other configuration options, such as the ports and network proxy to use, as shown in the following example. For more information, see Greengrass nucleus configuration.
>
> ```
> ---
> system:
>   rootpath: "/greengrass/v2"
>   # The following values are optional. Return them from the provisioning
>  plugin or set them here.
>   # certificateFilePath: ""
>   # privateKeyPath: ""
>   # rootCaPath: ""
>   # thingName: ""
> services:
>   aws.greengrass.Nucleus:
>     version: "2.18.4"
>     configuration:
>       mqtt:
>         port: 443
> ```

```
      greengrassDataPlanePort: 443
      networkProxy:
        noProxyAddresses: "http://192.168.0.1,www.example.com"
        proxy:
          url: "http://my-proxy-server:1100"
          username: "Mary_Major"
          password: "pass@word1357"
      # The following values are optional. Return them from the provisioning
 plugin or set them here.
      # awsRegion: ""
      # iotRoleAlias: ""
      # iotDataEndpoint: ""
      # iotCredEndpoint: ""
  com.example.CustomProvisioning:
    configuration:
      # You can specify configuration parameters to provide to your plugin.
      # pluginParameter: ""
```

3. Run the installer. Specify `--trusted-plugin` to provide your custom provisioning plugin, and specify `--init-config` to provide the configuration file.

> **Note**
>
> Windows has a path length limitation of 260 characters. If you are using Windows, use a root folder like `C:\greengrass\v2` or `D:\greengrass\v2` to keep the Greengrass components paths under the 260 character limit.

- Replace */greengrass/v2* or *C:\greengrass\v2* with the Greengrass root folder.
- Replace each instance of *GreengrassInstaller* with the folder where you unpacked the installer.
- Replace the path to the custom provisioning plugin JAR file with the path to your plugin's JAR file.

Linux or Unix

```
sudo -E java -Droot="/greengrass/v2" -Dlog.store=FILE \
  -jar ./GreengrassInstaller/lib/Greengrass.jar \
  --trusted-plugin /path/to/com.example.CustomProvisioning.jar \
```

```
  --init-config ./GreengrassInstaller/config.yaml \
  --component-default-user ggc_user:ggc_group \
  --setup-system-service true
```

Windows Command Prompt (CMD)

```
java -Droot="C:\greengrass\v2" "-Dlog.store=FILE" ^
  -jar ./GreengrassInstaller/lib/Greengrass.jar ^
  --trusted-plugin /path/to/com.example.CustomProvisioning.jar ^
  --init-config ./GreengrassInstaller/config.yaml ^
  --component-default-user ggc_user ^
  --setup-system-service true
```

PowerShell

```
java -Droot="C:\greengrass\v2" "-Dlog.store=FILE" `
  -jar ./GreengrassInstaller/lib/Greengrass.jar `
  --trusted-plugin /path/to/com.example.CustomProvisioning.jar `
  --init-config ./GreengrassInstaller/config.yaml `
  --component-default-user ggc_user `
  --setup-system-service true
```

Important

On Windows core devices, you must specify `--setup-system-service true` to set up the AWS IoT Greengrass Core software as a system service.

If you specify `--setup-system-service true`, the installer prints `Successfully set up Nucleus as a system service` if it set up and ran the software as a system service. Otherwise, the installer doesn't output any message if it installs the software successfully.

Note

You can't use the `deploy-dev-tools` argument to deploy local development tools when you run the installer without the `--provision true` argument. For information about deploying the Greengrass CLI directly on your device, see Greengrass Command Line Interface.

4. Verify the installation by viewing the files in the root folder.

   Linux or Unix

   ```
   ls /greengrass/v2
   ```

   Windows Command Prompt (CMD)

   ```
   dir C:\greengrass\v2
   ```

   PowerShell

   ```
   ls C:\greengrass\v2
   ```

   If the installation succeeded, the root folder contains several folders, such as `config`, `packages`, and `logs`.

If you installed the AWS IoT Greengrass Core software as a system service, the installer runs the software for you. Otherwise, you must run the software manually. For more information, see Run the AWS IoT Greengrass Core software.

For more information about how to configure and use the software and AWS IoT Greengrass, see the following:

- Configure the AWS IoT Greengrass Core software
- Develop AWS IoT Greengrass components
- Deploy AWS IoT Greengrass components to devices
- Greengrass Command Line Interface

Develop custom provisioning plugins

To develop a custom provisioning plugin, create a Java class that implements the `com.aws.greengrass.provisioning.DeviceIdentityInterface` interface. You can include the Greengrass nucleus JAR file in your project to access this interface and its classes. This interface defines a method that inputs a plugin configuration and outputs a provisioning configuration. The provisioning configuration defines configurations for the system and the Greengrass nucleus

component. The AWS IoT Greengrass Core software installer uses this provisioning configuration to configure the AWS IoT Greengrass Core software on a device.

After you develop a custom provisioning plugin, build it as a JAR file that you can provide to the AWS IoT Greengrass Core software installer to run your plugin during installation. The installer runs your custom provisioning plugin in the same JVM that the installer uses, so you can create a JAR that contains only your plugin code.

> **Note**
>
> The AWS IoT fleet provisioning plugin implements the `DeviceIdentityInterface` to use fleet provisioning during installation. The fleet provisioning plugin is open source, so you can explore its source code to see an example of how to use the provisioning plugin interface. For more information, see the AWS IoT fleet provisioning plugin on GitHub.

Topics

- Requirements
- Implement the DeviceIdentityInterface interface

Requirements

To develop a custom provisioning plugin, you must create a Java class that meets the following requirements:

- Uses the `com.aws.greengrass` package, or a package within the `com.aws.greengrass` package.
- Has a constructor without any arguments.
- Implements the `DeviceIdentityInterface` interface. For more information, see Implement the DeviceIdentityInterface interface.

Implement the DeviceIdentityInterface interface

To use the `com.aws.greengrass.provisioning.DeviceIdentityInterface` interface in your custom plugin, add the Greengrass nucleus as a dependency to your project.

To use the DeviceIdentityInterface in a custom provisioning plugin project

- You can add the Greengrass nucleus JAR file as a library, or add the Greengrass nucleus as a Maven dependency. Do one of the following:
  - To add the Greengrass nucleus JAR file as a library, download the AWS IoT Greengrass Core software, which contains the Greengrass nucleus JAR. You can download the latest version of the AWS IoT Greengrass Core software from the following location:
    - https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip

    You can find the Greengrass nucleus JAR file (`Greengrass.jar`) in the `lib` folder in the ZIP file. Add this JAR file to your project.
  - To consume the Greengrass nucleus in a Maven project, add a dependency the on the `nucleus` artifact in the `com.aws.greengrass` group. You must also add the `greengrass-common` repository, because the Greengrass nucleus isn't available in the Maven Central Repository.

```
<project ...>
    ...
    <repositories>
        <repository>
            <id>greengrass-common</id>
            <name>greengrass common</name>
            <url>https://d2jrmugq4soldf.cloudfront.net/snapshots</url>
        </repository>
    </repositories>
    ...
    <dependencies>
        <dependency>
            <groupId>com.aws.greengrass</groupId>
            <artifactId>nucleus</artifactId>
            <version>2.5.0-SNAPSHOT</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
</project>
```

The DeviceIdentityInterface interface

The `com.aws.greengrass.provisioning.DeviceIdentityInterface` interface has the following shape.

> **Note**
>
> You can also explore these classes in the com.aws.greengrass.provisioning package of the Greengrass nucleus source code on GitHub.

```
public interface com.aws.greengrass.provisioning.DeviceIdentityInterface {
    ProvisionConfiguration updateIdentityConfiguration(ProvisionContext context)
            throws RetryableProvisioningException, InterruptedException;

    // Return the name of the plugin.
    String name();
}

com.aws.greengrass.provisioning.ProvisionConfiguration {
    SystemConfiguration systemConfiguration;
    NucleusConfiguration nucleusConfiguration
}

com.aws.greengrass.provisioning.ProvisionConfiguration.SystemConfiguration {
    String certificateFilePath;
    String privateKeyPath;
    String rootCAPath;
    String thingName;
}

com.aws.greengrass.provisioning.ProvisionConfiguration.NucleusConfiguration {
    String awsRegion;
    String iotCredentialsEndpoint;
    String iotDataEndpoint;
    String iotRoleAlias;
}

com.aws.greengrass.provisioning.ProvisioningContext {
    Map<String, Object> parameterMap;
    String provisioningPolicy;  // The policy is always "PROVISION_IF_NOT_PROVISIONED".
}
```

```
com.aws.greengrass.provisioning.exceptions.RetryableProvisioningException {}
```

Each configuration value in the `SystemConfiguration` and `NucleusConfiguration` is required to install the AWS IoT Greengrass Core software, but you can return `null`. If your custom provisioning plugin returns `null` for any configuration value, you must provide that value in the system or nucleus configuration when you create the `config.yaml` file to provide to the AWS IoT Greengrass Core software installer. If your custom provisioning plugin returns a non-null value for an option that you also define in `config.yaml`, then the installer replaces the value in `config.yaml` with the value returned by the plugin.

Installer arguments

The AWS IoT Greengrass Core software includes an installer that sets up the software and provisions the required AWS resources for the Greengrass core device to run. The installer includes the following arguments that you can specify to configure the installation:

`-h, --help`

(Optional) Show the installer's help information.

`--version`

(Optional) Show the version of the AWS IoT Greengrass Core software.

`-Droot`

(Optional) The path to the folder to use as the root for the AWS IoT Greengrass Core software.

> **Note**
>
> This argument sets a JVM property, so you must specify it before `-jar` when you run the installer. For example, specify `java -Droot="/greengrass/v2" -jar /path/to/Greengrass.jar`.

Default:

- Linux: `~/.greengrass`
- Windows: `%USERPROFILE%/.greengrass`

`-ar`, `--aws-region`

The AWS Region that the AWS IoT Greengrass Core software uses to retrieve or create its required AWS resources.

`-p`, `--provision`

(Optional) You can register this device as an AWS IoT thing and provision the AWS resources that the core device requires. If you specify `true`, the AWS IoT Greengrass Core software provisions an AWS IoT thing, (optional) an AWS IoT thing group, an IAM role, and an AWS IoT role alias.

Default: `false`

`-tn`, `--thing-name`

(Optional) The name of the AWS IoT thing that you register as this core device. If the thing with the name doesn't exist in your AWS account, the AWS IoT Greengrass Core software creates it.

> **Note**
>
> The thing name can't contain colon (`:`) characters.

You must specify `--provision true` to apply this argument.

Default: `GreengrassV2IotThing_` plus a random UUID.

`-tgn`, `--thing-group-name`

(Optional) The name of the AWS IoT thing group where you add this core device's AWS IoT thing. If a deployment targets this thing group, this core device receives that deployment when it connects to AWS IoT Greengrass. If the thing group with this name doesn't exist in your AWS account, the AWS IoT Greengrass Core software creates it.

> **Note**
>
> The thing group name can't contain colon (`:`) characters.

You must specify `--provision true` to apply this argument.

`-tpn`, `--thing-policy-name`

This feature is available for v2.4.0 and later of the Greengrass nucleus component.

(Optional) The name of the AWS IoT policy to attach to this core device's AWS IoT thing certificate. If the AWS IoT policy with this name doesn't exist in your AWS account, the AWS IoT Greengrass Core software creates it.

The AWS IoT Greengrass Core software creates a permissive AWS IoT policy by default. You can scope down this policy, or create a custom policy where you restrict permissions for your use case. For more information, see Minimal AWS IoT policy for AWS IoT Greengrass V2 core devices.

You must specify `--provision true` to apply this argument.

Default: `GreengrassV2IoTThingPolicy`

`-trn`, `--tes-role-name`

(Optional) The name of the IAM role to use to acquire AWS credentials that let the core device interact with AWS services. If the role with this name doesn't exist in your AWS account, the AWS IoT Greengrass Core software creates it with the `GreengrassV2TokenExchangeRoleAccess` policy. This role doesn't have access to your S3 buckets where you host component artifacts. So, you must add permissions to your artifacts' S3 buckets and objects when you create a component. For more information, see Authorize core devices to interact with AWS services.

You must specify `--provision true` to apply this argument.

Default: `GreengrassV2TokenExchangeRole`

`-tra`, `--tes-role-alias-name`

(Optional) The name of the AWS IoT role alias that points to the IAM role that provides AWS credentials for this core device. If the role alias with this name doesn't exist in your AWS account, the AWS IoT Greengrass Core software creates it and points it to the IAM role that you specify.

You must specify `--provision true` to apply this argument.

Default: `GreengrassV2TokenExchangeRoleAlias`

`-ss, --setup-system-service`

(Optional) You can set up the AWS IoT Greengrass Core software as a system service that runs when this device boots. The system service name is `greengrass`. For more information, see Configure the Greengrass nucleus as a system service.

On Linux operating systems, this argument requires that the systemd init system is available on the device.

> **Important**
>
> On Windows core devices, you must set up the AWS IoT Greengrass Core software as a system service.

Default: `false`

`-u, --component-default-user`

The name or ID of the user that the AWS IoT Greengrass Core software uses to run components. For example, you can specify **`ggc_user`**. This value is required when you run the installer on Windows operating systems.

On Linux operating systems, you can also optionally specify the group. Specify the user and group separated by a colon. For example, **`ggc_user:ggc_group`**.

The following additional considerations apply for Linux operating systems:

- If you run as root, the default component user is the user that is defined in the configuration file. If the configuration file doesn't define a user, this defaults to `ggc_user:ggc_group`. If `ggc_user` or `ggc_group` don't exist, the software creates them.
- If you run as a non-root user, the AWS IoT Greengrass Core software uses that user to run components.
- If you don't specify a group, the AWS IoT Greengrass Core software uses the primary group of the system user.

For more information, see Configure the user that runs components.

`-d, --deploy-dev-tools`

(Optional) You can download and deploy the Greengrass CLI component to this core device. You can use this tool to develop and debug components on this core device.

> **Important**
>
> We recommend that you use this component in only development environments, not production environments. This component provides access to information and operations that you typically won't need in a production environment. Follow the principle of least privilege by deploying this component to only core devices where you need it.

You must specify `--provision true` to apply this argument.

Default: `false`

`-init`, `--init-config`

(Optional) The path to the configuration file to use to install the AWS IoT Greengrass Core software. You can use this option to set up new core devices with a specific nucleus configuration, for example.

> **Important**
>
> The configuration file that you specify merges with the existing configuration file on the core device. This includes the components and component configurations on the core device. We recommend the configuration file only lists the configurations that you are trying to change.

`-tp`, `--trusted-plugin`

(Optional) The path to a JAR file to load as a trusted plugin. Use this option to provide provisioning plugin JAR files, such as to install with fleet provisioning or custom provisioning, or to install with the private key and certificate in a hardware security module.

`-s`, `--start`

(Optional) You can start the AWS IoT Greengrass Core software after it installs and, optionally, provisions resources.

Default: `true`

Run the AWS IoT Greengrass Core software

After you install the AWS IoT Greengrass Core software, run it to connect your device to AWS IoT Greengrass.

When you install the AWS IoT Greengrass Core software, you can specify whether to install it as a system service with systemd. If you choose this option, the installer runs the software for you and configures it to run when your device boots.

> **Important**
>
> On Windows core devices, you must set up the AWS IoT Greengrass Core software as a system service.

Topics

- Check if the AWS IoT Greengrass Core software runs as a system service
- Run the AWS IoT Greengrass Core software as a system service
- Run the AWS IoT Greengrass Core software without a system service

Check if the AWS IoT Greengrass Core software runs as a system service

When you install the AWS IoT Greengrass Core software, you can specify the `--setup-system-service true` argument to install the AWS IoT Greengrass Core software as a system service. Linux devices require the systemd init system to set up the AWS IoT Greengrass Core software as a system service. If you use this option, the installer runs the software for you and configures it to run when your device boots. The installer outputs the following message if it successfully installs the AWS IoT Greengrass Core software as a system service.

```
Successfully set up Nucleus as a system service
```

If you previously installed the AWS IoT Greengrass Core software and don't have the installer output, you can check if the software installed as a system service.

To check if the AWS IoT Greengrass Core software is installed as a system service

- Run the following command to check the status of the Greengrass system service.

Linux or Unix (systemd)

```
sudo systemctl status greengrass.service
```

The response looks similar to the following example if the AWS IoT Greengrass Core software is installed as a system service and active.

```
# greengrass.service - Greengrass Core
   Loaded: loaded (/etc/systemd/system/greengrass.service; enabled; vendor
 preset: disabled)
   Active: active (running) since Thu 2021-02-11 01:33:44 UTC; 4 days ago
 Main PID: 16107 (sh)
   CGroup: /system.slice/greengrass.service
           ##16107 /bin/sh /greengrass/v2/alts/current/distro/bin/loader
           ##16111 java -Dlog.store=FILE -Droot=/greengrass/v2 -jar /greengrass/
v2/alts/current/distro/lib/Greengrass...
```

If `systemctl` or `greengrass.service` isn't found, the AWS IoT Greengrass Core software isn't installed as a system service. To run the software, see [Run the AWS IoT Greengrass Core software without a system service].

Windows Command Prompt (CMD)

```
sc query greengrass
```

The response looks similar to the following example if the AWS IoT Greengrass Core software is installed as a Windows service and active.

```
SERVICE_NAME: greengrass
        TYPE               : 10  WIN32_OWN_PROCESS
        STATE              : 4  RUNNING
                                (STOPPABLE, NOT_PAUSABLE, ACCEPTS_SHUTDOWN)
        WIN32_EXIT_CODE    : 0  (0x0)
        SERVICE_EXIT_CODE  : 0  (0x0)
        CHECKPOINT         : 0x0
        WAIT_HINT          : 0x0
```

PowerShell

```
Get-Service greengrass
```

The response looks similar to the following example if the AWS IoT Greengrass Core software is installed as a Windows service and active.

```
Status   Name               DisplayName
------   ----               -----------
Running  greengrass         greengrass
```

Run the AWS IoT Greengrass Core software as a system service

If the AWS IoT Greengrass Core software is installed as a system service, you can use the system service manager to start, stop, and manage the software. For more information, see Configure the Greengrass nucleus as a system service.

To run the AWS IoT Greengrass Core software

- Run the following command to start the AWS IoT Greengrass Core software.

  Linux or Unix (systemd)

  ```
  sudo systemctl start greengrass.service
  ```

  Windows Command Prompt (CMD)

  ```
  sc start greengrass
  ```

  PowerShell

  ```
  Start-Service greengrass
  ```

Run the AWS IoT Greengrass Core software without a system service

On Linux core devices, if the AWS IoT Greengrass Core software isn't installed as a system service, you can run the software's loader script to run the software.

To run the AWS IoT Greengrass Core software without a system service

- Run the following command to start the AWS IoT Greengrass Core software. If you run this command in a terminal, you must keep the terminal session open to keep the AWS IoT Greengrass Core software running.
  - Replace */greengrass/v2* or *C:\greengrass\v2* with the Greengrass root folder that you use.

```
sudo /greengrass/v2/alts/current/distro/bin/loader
```

The software prints the following message if it launches successfully.

```
Launched Nucleus successfully.
```

Run AWS IoT Greengrass Core software in a Docker container

AWS IoT Greengrass can be configured to run in a Docker container. Docker is a platform that provides the tools for you to build, run, test, and deploy applications that are based on Linux containers. When you run an AWS IoT Greengrass Docker image, you can choose whether to provide your AWS credentials to the Docker container and allow the AWS IoT Greengrass Core software installer to automatically provision the AWS resources that a Greengrass core device requires to operate. If you don't want to provide AWS credentials, then you can manually provision AWS resources and run AWS IoT Greengrass Core software in the Docker container.

Topics

- Supported platforms and requirements
- AWS IoT Greengrass Docker software downloads
- Choose how to provision AWS resources
- Build the AWS IoT Greengrass container image from a Dockerfile
- Run AWS IoT Greengrass in a Docker container with automatic resource provisioning
- Run AWS IoT Greengrass in a Docker container with manual resource provisioning
- Troubleshooting AWS IoT Greengrass in a Docker container

Supported platforms and requirements

Host computers must meet the following minimum requirements to install and run the AWS IoT Greengrass Core software in a Docker container:

- A Linux-based operating system with an internet connection.
- Docker Engine version 18.09 or later.
- (Optional) Docker Compose version 1.22 or later. Docker Compose is required only if you want to use the Docker Compose CLI to run your Docker images.

To run Lambda function components inside of the Docker container, you must configure the container to meet additional requirements. For more information, see Lambda function requirements.

Run components in process mode

AWS IoT Greengrass doesn't support running Lambda functions or AWS-provided components in an isolated runtime environment inside the AWS IoT Greengrass Docker container. You must run these components in process mode without any isolation.

When you configure a Lambda function component, set the isolation mode to **No container**. For more information, see Run AWS Lambda functions.

When you deploy any of the following AWS-provided components, update the configuration for each component to set the `containerMode` parameter to `NoContainer`. For more information about configuration updates, see Update component configurations.

- CloudWatch metrics
- Device Defender
- Firehose
- Modbus-RTU protocol adapter
- Amazon SNS

AWS IoT Greengrass Docker software downloads

AWS IoT Greengrass provides a Dockerfile to build a container image that has AWS IoT Greengrass Core software and dependencies installed on an Amazon Linux 2 (x86_64) base image. You can

modify the base image in the Dockerfile to run AWS IoT Greengrass on a different platform architecture.

Download the Dockerfile package from GitHub.

The Dockerfile uses an older version of Greengrass. You should update the file to use the version of Greengrass that you want. For information about building the AWS IoT Greengrass container image from the Dockerfile, see Build the AWS IoT Greengrass container image from a Dockerfile.

Choose how to provision AWS resources

When you install the AWS IoT Greengrass Core software in a Docker container, you can choose whether to automatically provision the AWS resources that a Greengrass core device requires to operate, or to use resources that you manually provision.

- **Automatic resource provisioning**—The installer provisions the AWS IoT thing, AWS IoT thing group, IAM role, and AWS IoT role alias when you run the AWS IoT Greengrass container image for the first time. The installer can also deploy the local development tools to the core device, so you can use the device to develop and test custom software components. To automatically provision these resources, you must provide AWS credentials as environment variables to the Docker image.

  To use automatic provisioning, you must set the Docker environment variable `PROVISION=true` and mount a credential file to provide your AWS credentials to the container.
- **Manual resource provisioning**—If you don't want to provide AWS credentials to the container, then you can manually provision the AWS resources before you run the AWS IoT Greengrass container image. You must create a configuration file to provide information about these resources to the AWS IoT Greengrass Core software installer within the Docker container.

  To use manual provisioning, you must set the Docker environment variable `PROVISION=false`. Manual provisioning is the default option.

For more information, see Build the AWS IoT Greengrass container image from a Dockerfile.

Build the AWS IoT Greengrass container image from a Dockerfile

AWS provides a Dockerfile that you can download and use to run AWS IoT Greengrass Core software in a Docker container. Dockerfiles contain source code for building AWS IoT Greengrass container images.

Before you build an AWS IoT Greengrass container image, you must configure your Dockerfile to select the version of AWS IoT Greengrass Core software that you want to install. You can also configure environment variables to choose how to provision resources during installation, and customize other installation options. This section describes how to configure and build an AWS IoT Greengrass Docker image from a Dockerfile.

Download the Dockerfile package

You can download the AWS IoT Greengrass Dockerfile package from GitHub:

AWS Greengrass Docker Repository

After you download the package, extract the contents to the *download-directory/*aws-greengrass-docker-*nucleus-version* folder on your computer. The Dockerfile uses an older version of Greengrass. You should update the file to use the version of Greengrass that you want.

Specify the AWS IoT Greengrass Core software version

Use the following build argument in the Dockerfile to specify the version of the AWS IoT Greengrass Core software that you want to use in the AWS IoT Greengrass Docker image. By default, the Dockerfile uses the latest version of the AWS IoT Greengrass Core software.

`GREENGRASS_RELEASE_VERSION`

The version of the AWS IoT Greengrass Core software. By default, the Dockerfile downloads the latest available version of the Greengrass nucleus. Set the value to the version of the nucleus that you want to download.

Set environment variables

Environment variables enable you to customize how AWS IoT Greengrass Core software is installed in the Docker container. You can set environment variables for your AWS IoT Greengrass Docker image in various ways.

- To use the same environment variables to create multiple images, set environment variables directly in the Dockerfile.
- If you use `docker run` to start your container, pass environment variables as arguments in the command, or set environment variables in an environment variables file and then pass the file

as an argument. For more information about setting environment variables in Docker, see the environment variables in the Docker documentation.

- If you use `docker-compose up` to start your container, set environment variables in an environment variables file and then pass the file as an argument. For more information about setting environment variables in Compose, see the Docker documentation.

You can configure the following environment variables for the AWS IoT Greengrass Docker image.

> **Note**
>
> Don't modify the `TINI_KILL_PROCESS_GROUP` variable in the Dockerfile. This variable allows forwarding `SIGTERM` to all PIDs in the PID group so that AWS IoT Greengrass Core software can shut down correctly when the Docker container is stopped.

`GGC_ROOT_PATH`

(Optional) The path to the folder within the container to use as the root for AWS IoT Greengrass Core software.

Default: `/greengrass/v2`

`PROVISION`

(Optional) Determines whether the AWS IoT Greengrass Core provisions AWS resources.

- If you specify `true`, AWS IoT Greengrass Core software registers the container image as an AWS IoT thing and provisions the AWS resources that the Greengrass core device requires. The AWS IoT Greengrass Core software provisions an AWS IoT thing, (optional) an AWS IoT thing group, an IAM role, and an AWS IoT role alias. For more information, see Run AWS IoT Greengrass in a Docker container with automatic resource provisioning.
- If you specify `false`, then you must create a configuration file to provide to the AWS IoT Greengrass Core installer that specifies to use the AWS resources and certificates that you manually created. For more information, see Run AWS IoT Greengrass in a Docker container with manual resource provisioning.

Default: `false`

`AWS_REGION`

(Optional) The AWS Region that the AWS IoT Greengrass Core software uses to retrieve or create required AWS resources.

Default: `us-east-1`.

`THING_NAME`

(Optional) The name of the AWS IoT thing that you register as this core device. If the thing with this name doesn't exist in your AWS account, the AWS IoT Greengrass Core software creates it.

You must specify `PROVISION=true` to apply this argument.

Default: `GreengrassV2IotThing_` plus a random UUID.

`THING_GROUP_NAME`

(Optional) The name of the AWS IoT thing group where you add this core device's AWS IoT If a deployment targets this thing group, this and other core devices in that group receive that deployment when it connects to AWS IoT Greengrass. If the thing group with this name doesn't exist in your AWS account, the AWS IoT Greengrass Core software creates it.

You must specify `PROVISION=true` to apply this argument.

`TES_ROLE_NAME`

(Optional) The name of the IAM role to use to acquire AWS credentials that let the Greengrass core device interact with AWS services. If the role with this name doesn't exist in your AWS account, the AWS IoT Greengrass Core software creates it with the `GreengrassV2TokenExchangeRoleAccess` policy. This role doesn't have access to your S3 buckets where you host component artifacts. So, you must add permissions to your artifacts' S3 buckets and objects when you create a component. For more information, see Authorize core devices to interact with AWS services.

Default: `GreengrassV2TokenExchangeRole`

`TES_ROLE_ALIAS_NAME`

(Optional) The name of the AWS IoT role alias that points to the IAM role that provides AWS credentials for the Greengrass core device. If the role alias with this name doesn't exist in your AWS account, the AWS IoT Greengrass Core software creates it and points it to the IAM role that you specify.

Default: `GreengrassV2TokenExchangeRoleAlias`

`COMPONENT_DEFAULT_USER`

(Optional) The name or ID of the system user and group that the AWS IoT Greengrass Core software uses to run components. Specify the user and group, separated by a colon. The group is optional. For example, you can specify **`ggc_user:ggc_group`** or **`ggc_user`**.

- If you run as root, this defaults to the user and group that the configuration file defines. If the configuration file doesn't define a user and group, this defaults to `ggc_user:ggc_group`. If `ggc_user` or `ggc_group` don't exist, the software creates them.
- If you run as a non-root user, the AWS IoT Greengrass Core software uses that user to run components.
- If you don't specify a group, the AWS IoT Greengrass Core software uses the primary group of the system user.

For more information, see Configure the user that runs components.

`DEPLOY_DEV_TOOLS`

Defines whether to download and deploy the Greengrass CLI component in the container image. You can use the Greengrass CLI to develop and debug components locally.

> **Important**
>
> We recommend that you use this component in only development environments, not production environments. This component provides access to information and operations that you typically won't need in a production environment. Follow the principle of least privilege by deploying this component to only core devices where you need it.

Default: `false`

`INIT_CONFIG`

(Optional) The path to the configuration file to use to install the AWS IoT Greengrass Core software. You can use this option to set up new Greengrass core devices with a specific nucleus configuration, or to specify manually provisioned resources, for example. You must mount your configuration file to the path that you specify in this argument.

`TRUSTED_PLUGIN`

This feature is available for v2.4.0 and later of the Greengrass nucleus component.

(Optional) The path to a JAR file to load as a trusted plugin. Use this option to provide provisioning plugin JAR files, such as to install with fleet provisioning or custom provisioning.

`THING_POLICY_NAME`

This feature is available for v2.4.0 and later of the Greengrass nucleus component.

(Optional) The name of the AWS IoT policy to attach to this core device's AWS IoT thing certificate. If the AWS IoT policy with this name doesn't exist in your AWS account the AWS IoT Greengrass Core software creates it.

You must specify `PROVISION=true` to apply this argument.

> **Note**
>
> The AWS IoT Greengrass Core software creates a permissive AWS IoT policy by default. You can scope down this policy, or create a custom policy where you restrict permissions for your use case. For more information, see Minimal AWS IoT policy for AWS IoT Greengrass V2 core devices.

Specify the dependencies to install

The RUN instruction in the AWS IoT Greengrass Dockerfile prepares up the container environment to run the AWS IoT Greengrass Core software installer. You can customize the dependencies that are installed before the AWS IoT Greengrass Core software installer runs in the Docker container.

Build the AWS IoT Greengrass image

Use the AWS IoT Greengrass Dockerfile to build an AWS IoT Greengrass container image. You can use the Docker CLI or the Docker Compose CLI to build the image and start the container. You can also use the Docker CLI to build the image and then use Docker Compose to start your container from that image.

Docker

1. On the host machine, run the following command to switch to the directory that contains the configured Dockerfile.

   ```
   cd download-directory/aws-greengrass-docker-nucleus-version
   ```

2. Run the following command to build the AWS IoT Greengrass container image from the Dockerfile.

```
sudo docker build -t "platform/aws-iot-greengrass:nucleus-version" ./
```

Docker Compose

1. On the host machine, run the following command to switch to the directory that contains the Dockerfile and the Compose file.

```
cd download-directory/aws-greengrass-docker-nucleus-version
```

2. Run the following command to use the Compose file to build the AWS IoT Greengrass container image.

```
docker-compose -f docker-compose.yml build
```

You have successfully created the AWS IoT Greengrass container image. The Docker image has the AWS IoT Greengrass Core software installed. You can now run the AWS IoT Greengrass Core software in a Docker container.

Run AWS IoT Greengrass in a Docker container with automatic resource provisioning

This tutorial shows you how to install and run AWS IoT Greengrass Core software in a Docker container with automatically provisioned AWS resources and local development tools. You can use this development environment to explore AWS IoT Greengrass features in a Docker container. The software requires AWS credentials to provision these resources and deploy the local development tools.

If you can't provide AWS credentials to the container, you can provision the AWS resources that the core device requires to operate. You can also deploy the development tools to a core device to use as a development device. This enables you to provide fewer permissions to the device when you run the container. For more information, see Run AWS IoT Greengrass in a Docker container with manual resource provisioning.

Prerequisites

To complete this tutorial, you need the following.

- An AWS account. If you don't have one, see Set up an AWS account.
- An AWS IAM user with permissions to provision the AWS IoT and IAM resources for a Greengrass core device. The AWS IoT Greengrass Core software installer uses your AWS credentials to automatically provision these resources. For information about the minimal IAM policy to automatically provision resources, see Minimal IAM policy for installer to provision resources.
- An AWS IoT Greengrass Docker image. You can build an image from the AWS IoT Greengrass Dockerfile.
- The host computer where you run the Docker container must meet the following requirements:
  - A Linux-based operating system with an internet connection.
  - Docker Engine version 18.09 or later.
  - (Optional) Docker Compose version 1.22 or later. Docker Compose is required only if you want to use the Docker Compose CLI to run your Docker images.

Configure your AWS credentials

In this step, you create a credential file on the host computer that contains your AWS security credentials. When you run the AWS IoT Greengrass Docker image, you must mount the folder that contains this credential file to `/root/.aws/` in the Docker container. The AWS IoT Greengrass installer uses these credentials to provision resources in your AWS account. For information about the minimal IAM policy that the installer requires to automatically provision resources, see Minimal IAM policy for installer to provision resources.

1. Retrieve one of the following.

   - Long-term credentials for an IAM user. For information about how to retrieve long-term credentials, see Managing access keys for IAM users in the *IAM User Guide*.
   - (Recommended) Temporary credentials for an IAM role. For information about how to retrieve temporary credentials, see Using temporary security credentials with the AWS CLI in the *IAM User Guide*.

2. Create a folder where you place your credential file.

   ```
   mkdir ./greengrass-v2-credentials
   ```

3. Use a text editor to create a configuration file named `credentials` in the `./greengrass-v2-credentials` folder.

   For example, you can run the following command to use GNU nano to create the `credentials` file.

   ```
   nano ./greengrass-v2-credentials/credentials
   ```

4. Add your AWS credentials to the `credentials` file in the following format.

   ```
   [default]
   aws_access_key_id = AKIAIOSFODNN7EXAMPLE
   aws_secret_access_key = wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
   aws_session_token
    = AQoEXAMPLEH4aoAH0gNCAPy...truncated...zrkuWJOgQs8IZZaIv2BXIa2R4Olgk
   ```

   Include `aws_session_token` for temporary credentials only.

> **Important**
>
> Remove the credential file from the host computer after you start the AWS IoT Greengrass container. If you don't remove the credential file, then your AWS credentials will remain mounted inside the container. For more information, see Run the AWS IoT Greengrass Core software in a container.

Create an environment file

This tutorial uses an environment file to set the environment variables that will be passed to the AWS IoT Greengrass Core software installer inside the Docker container. You can also use the `-e` or `--env` argument in your `docker run` command to set environment variables in the Docker container or you can set the variables in an `environment` block in the `docker-compose.yml` file.

1. Use a text editor to create an environment file named `.env`.

   For example, on a Linux-based system, you can run the following command to use GNU nano to create the `.env` in the current directory.

```
nano .env
```

2. Copy the following content into the file.

```
GGC_ROOT_PATH=/greengrass/v2
AWS_REGION=region
PROVISION=true
THING_NAME=MyGreengrassCore
THING_GROUP_NAME=MyGreengrassCoreGroup
TES_ROLE_NAME=GreengrassV2TokenExchangeRole
TES_ROLE_ALIAS_NAME=GreengrassCoreTokenExchangeRoleAlias
COMPONENT_DEFAULT_USER=ggc_user:ggc_group
```

Then, replace the following values.

- */greengrass/v2*. The Greengrass root folder that you want to use for installation. You use the GGC_ROOT environment variable to set this value.
- *region*. The AWS Region where you created the resources.
- *MyGreengrassCore*. The name of the AWS IoT thing. If the thing doesn't exist, the installer creates it. The installer downloads the certificates to authenticate as the AWS IoT thing.
- *MyGreengrassCoreGroup*. The name of the AWS IoT thing group. If the thing group doesn't exist, the installer creates it and adds the thing to it. If the thing group exists and has an active deployment, the core device downloads and runs the software that the deployment specifies.
- *GreengrassV2TokenExchangeRole*. Replace with the name of the IAM token exchange role that allows the Greengrass core device to get temporary AWS credentials. If the role doesn't exist, the installer creates it and creates and attaches a policy named *GreengrassV2TokenExchangeRole*Access. For more information, see Authorize core devices to interact with AWS services.
- *GreengrassCoreTokenExchangeRoleAlias*. The token exchange role alias. If the role alias doesn't exist, the installer creates it and points it to the IAM token exchange role that you specify. For more information, see

> **Note**
>
> You can set the `DEPLOY_DEV_TOOLS` environment variable to `true` to deploy the Greengrass CLI component, which enables you to develop custom components inside of the Docker container. We recommend that you use this component in only development environments, not production environments. This component provides access to information and operations that you typically won't need in a production environment. Follow the principle of least privilege by deploying this component to only core devices where you need it.

Run the AWS IoT Greengrass Core software in a container

This tutorial shows you how to start the Docker image that you built in a the Docker container. You can use the Docker CLI or the Docker Compose CLI to run the AWS IoT Greengrass Core software image in a Docker container.

Docker

1. Run the following command to start the Docker container.

   ```
   docker run --rm --init -it --name docker-image \
    -v path/to/greengrass-v2-credentials:/root/.aws/:ro \
    --env-file .env \
    -p 8883 \
    your-container-image:version
   ```

   This example command uses the following arguments for docker run:

   - `--rm`. Cleans up the container when it exits.
   - `--init`. Uses an init process in the container.

     > **Note**
     >
     > The `--init` argument is required to shut down AWS IoT Greengrass Core software when you stop the Docker container.

- `-it`. (Optional) Runs the Docker container in the foreground as an interactive process. You can replace this with the `-d` argument to run the Docker container in detached mode instead. For more information, see Detached vs foreground in the Docker documentation.
- `--name`. Runs a container named `aws-iot-greengrass`
- `-v`. Mounts a volume into the Docker container to make the configuration file and the certificate files available to AWS IoT Greengrass running inside the container.
- `--env-file`. (Optional) Specifies the environment file to set the environment variables that will be passed to the AWS IoT Greengrass Core software installer inside the Docker container. This argument is required only if you created an environment file to set environment variables. If you didn't create an environment file, you can use `--env` arguments to set environment variables directly in your Docker run command.
- `-p`. (Optional) Publishes the 8883 container port to the host machine. This argument is required if you want to connect and communicate over MQTT because AWS IoT Greengrass uses port 8883 for MQTT traffic. To open other ports, use additional `-p` arguments.

> **Note**
>
> To run your Docker container with increased security, you can use the `--cap-drop` and `--cap-add` arguments to selectively enable Linux capabilities for your container. For more information, see Runtime privilege and Linux capabilities in the Docker documentation.

2. Remove the credentials from `./greengrass-v2-credentials` on the host device.

```
rm -rf ./greengrass-v2-credentials
```

> **Important**
>
> You're removing these credentials, because they provide broad permissions that the core device needs only during setup. If you don't remove these credentials, Greengrass components and other processes running in the container can access them. If you need to provide AWS credentials to a Greengrass component, use the token exchange service. For more information, see Interact with AWS services.

Docker Compose

1. Use a text editor to create a Docker Compose file named `docker-compose.yml`.

   For example, on a Linux-based system, you can run the following command to use GNU nano to create the `docker-compose.yml` in the current directory.

   ```
   nano docker-compose.yml
   ```

   > **Note**
   >
   > You can also download and use the latest version of the AWS-provided Compose file from GitHub.

2. Add the following content to the Compose file. Your file should look similar to the following example. Replace *`docker-image`* with the name of your Docker image.

   ```
   version: '3.7'

   services:
     greengrass:
       init: true
       container_name: aws-iot-greengrass
       image: docker-image
       volumes:
         - ./greengrass-v2-credentials:/root/.aws/:ro
       env_file: .env
       ports:
         - "8883:8883"
   ```

   The following parameters in this example Compose file are optional:

   - `ports`—Publishes the 8883 container ports to the host machine. This parameter is required if you want to connect and communicate over MQTT because AWS IoT Greengrass uses port 8883 for MQTT traffic.
   - `env_file`—Specifies the environment file to set the environment variables that will be passed to the AWS IoT Greengrass Core software installer inside the Docker container. This parameter is required only if you created an environment file to set environment

variables. If you didn't create an environment file, you can use the environment parameter to set the variables directly in your Compose file.

> **Note**
>
> To run your Docker container with increased security, you can use `cap_drop` and `cap_add` in your Compose file to selectively enable Linux capabilities for your container. For more information, see Runtime privilege and Linux capabilities in the Docker documentation.

3. Run the following command to start the Docker container.

```
docker-compose -f docker-compose.yml up
```

4. Remove the credentials from `./greengrass-v2-credentials` on the host device.

```
rm -rf ./greengrass-v2-credentials
```

> **Important**
>
> You're removing these credentials, because they provide broad permissions that the core device needs only during setup. If you don't remove these credentials, Greengrass components and other processes running in the container can access them. If you need to provide AWS credentials to a Greengrass component, use the token exchange service. For more information, see Interact with AWS services.

Next steps

AWS IoT Greengrass Core software is now running in a Docker container. Run the following command to retrieve the container ID for the currently running container.

```
docker ps
```

You can then run the following command to access the container and explore AWS IoT Greengrass Core software running inside the container.

```
docker exec -it container-id /bin/bash
```

For information about creating a simple component, see Step 4: Develop and test a component on your device in Tutorial: Getting started with AWS IoT Greengrass V2

> **Note**
>
> When you use `docker exec` to run commands inside the Docker container, those commands are not logged in the Docker logs. To log your commands in the Docker logs, attach an interactive shell to the Docker container. For more information, see Attach an interactive shell to the Docker container.

The AWS IoT Greengrass Core log file is called `greengrass.log` and is located in */greengrass/v2*`/logs`. Component log files are also located in the same directory. To copy Greengrass logs to a temporary directory on the host, run the following command:

```
docker cp container-id:/greengrass/v2/logs /tmp/logs
```

If you want to persist logs after a container exits or has been removed, we recommend that you bind-mount only the */greengrass/v2*`/logs` directory to the temporary logs directory on the host instead of mounting the entire Greengrass directory. For more information, see Persist Greengrass logs outside of the Docker container.

To stop a running AWS IoT Greengrass Docker container, run `docker stop` or `docker-compose -f docker-compose.yml stop`. This action sends `SIGTERM` to the Greengrass process and shuts down all associated processes that were started in the container. The Docker container is initialized with the `docker-init` executable as process PID 1, which helps in removing any leftover zombie processes. For more information, see the Specify an init process in the Docker documentation.

For information about troubleshooting issues with running AWS IoT Greengrass in a Docker container, see Troubleshooting AWS IoT Greengrass in a Docker container.

Run AWS IoT Greengrass in a Docker container with manual resource provisioning

This tutorial shows you how to install and run AWS IoT Greengrass Core software in Docker container with manually provisioned AWS resources.

Topics

- Prerequisites
- Retrieve AWS IoT endpoints
- Create an AWS IoT thing
- Create the thing certificate
- Create a token exchange role
- Download certificates to the device
- Create a configuration file
- Create an environment file
- Run the AWS IoT Greengrass Core software in a container
- Next steps

Prerequisites

To complete this tutorial, you need the following:

- An AWS account. If you don't have one, see Set up an AWS account.
- An AWS IoT Greengrass Docker image. You can build an image from the AWS IoT Greengrass Dockerfile.
- The host computer where you run the Docker container must meet the following requirements:
  - A Linux-based operating system with an internet connection.
  - Docker Engine version 18.09 or later.
  - (Optional) Docker Compose version 1.22 or later. Docker Compose is required only if you want to use the Docker Compose CLI to run your Docker images.

Retrieve AWS IoT endpoints

Get the AWS IoT endpoints for your AWS account, and save them to use later. Your device uses these endpoints to connect to AWS IoT. Do the following:

1. Get the AWS IoT data endpoint for your AWS account.

```
aws iot describe-endpoint --endpoint-type iot:Data-ATS
```

The response looks similar to the following example, if the request succeeds.

```
{
  "endpointAddress": "device-data-prefix-ats.iot.us-west-2.amazonaws.com"
}
```

2. Get the AWS IoT credentials endpoint for your AWS account.

```
aws iot describe-endpoint --endpoint-type iot:CredentialProvider
```

The response looks similar to the following example, if the request succeeds.

```
{
  "endpointAddress": "device-credentials-prefix.credentials.iot.us-
west-2.amazonaws.com"
}
```

Create an AWS IoT thing

AWS IoT *things* represent devices and logical entities that connect to AWS IoT. Greengrass core devices are AWS IoT things. When you register a device as an AWS IoT thing, that device can use a digital certificate to authenticate with AWS.

In this section, you create an AWS IoT thing that represents your device.

To create an AWS IoT thing

1. Create an AWS IoT thing for your device. On your development computer, run the following command.

   - Replace *MyGreengrassCore* with the thing name to use. This name is also the name of your Greengrass core device.

     > **Note**
     >
     > The thing name can't contain colon (:) characters.

```
aws iot create-thing --thing-name MyGreengrassCore
```

The response looks similar to the following example, if the request succeeds.

```
{
  "thingName": "MyGreengrassCore",
  "thingArn": "arn:aws:iot:us-west-2:123456789012:thing/MyGreengrassCore",
  "thingId": "8cb4b6cd-268e-495d-b5b9-1713d71dbf42"
}
```

2. (Optional) Add the AWS IoT thing to a new or existing thing group. You use thing groups to manage fleets of Greengrass core devices. When you deploy software components to your devices, you can target individual devices or groups of devices. You can add a device to a thing group with an active Greengrass deployment to deploy that thing group's software components to the device. Do the following:

   a. (Optional) Create an AWS IoT thing group.

      - Replace *MyGreengrassCoreGroup* with the name of the thing group to create.

      > **Note**
      >
      > The thing group name can't contain colon (`:`) characters.

      ```
      aws iot create-thing-group --thing-group-name MyGreengrassCoreGroup
      ```

      The response looks similar to the following example, if the request succeeds.

      ```
      {
        "thingGroupName": "MyGreengrassCoreGroup",
        "thingGroupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/
      MyGreengrassCoreGroup",
        "thingGroupId": "4df721e1-ff9f-4f97-92dd-02db4e3f03aa"
      }
      ```

   b. Add the AWS IoT thing to a thing group.

- Replace *MyGreengrassCore* with the name of your AWS IoT thing.
- Replace *MyGreengrassCoreGroup* with the name of the thing group.

```
aws iot add-thing-to-thing-group --thing-name MyGreengrassCore --thing-group-
name MyGreengrassCoreGroup
```

The command doesn't have any output if the request succeeds.

Create the thing certificate

When you register a device as an AWS IoT thing, that device can use a digital certificate to authenticate with AWS. This certificate allows the device to communicate with AWS IoT and AWS IoT Greengrass.

In this section, you create and download certificates that your device can use to connect to AWS.

To create the thing certificate

1. Create a folder where you download the certificates for the AWS IoT thing.

   ```
   mkdir greengrass-v2-certs
   ```

2. Create and download the certificates for the AWS IoT thing.

   ```
   aws iot create-keys-and-certificate --set-as-active --certificate-pem-outfile
    greengrass-v2-certs/device.pem.crt --public-key-outfile greengrass-v2-certs/
   public.pem.key --private-key-outfile greengrass-v2-certs/private.pem.key
   ```

   The response looks similar to the following example, if the request succeeds.

   ```
   {
     "certificateArn": "arn:aws:iot:us-west-2:123456789012:cert/
   aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4",
     "certificateId":
    "aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4",
     "certificatePem": "-----BEGIN CERTIFICATE-----
   MIICiTCCAfICCQD6m7oRw0uXOjANBgkqhkiG9w
    0BAQUFADCBiDELMAkGA1UEBhMCVVMxCzAJBgNVBAgTAldBMRAwDgYDVQQHEwdTZ
   ```

```
 WF0dGxlMQ8wDQYDVQQKEwZBbWF6b24xFDASBgNVBAsTC0lBTSBDb25zb2xlMRIw
 EAYDVQQDEwlUZXN0Q2lsYWMxHzAdBgkqhkiG9w0BCQEWEG5vb25lQGFtYXpvbi5
 jb20wHhcNMTEwNDI1MjA0NTIxWhcNMTIwNDI0MjA0NTIxWjCBiDELMAkGA1UEBh
 MCVVMxCzAJBgNVBAgTAldBMRAwDgYDVQQHEwdTZWF0dGxlMQ8wDQYDVQQKEwZBb
 WF6b24xFDASBgNVBAsTC0lBTSBDb25zb2xlMRIwEAYDVQQDEwlUZXN0Q2lsYWMx
 HzAdBgkqhkiG9w0BCQEWEG5vb25lQGFtYXpvbi5jb20wgZ8wDQYJKoZIhvcNAQE
 BBQADgY0AMIGJAoGBAMaK0dn+a4GmWIWJ21uUSfwfEvySWtC2XADZ4nB+BLYgVI
 k60CpiwsZ3G93vUEIO3IyNoH/f0wYK8m9TrDHudUZg3qX4waLG5M43q7Wgc/MbQ
 ITxOUSQv7c7ugFFDzQGBzZswY6786m86gpEIbb3OhjZnzcvQAaRHhdlQWIMm2nr
 AgMBAAEwDQYJKoZIhvcNAQEFBQADgYEAtCu4nUhVVxYUntneD9+h8Mg9q6q+auN
 KyExzyLwaxlAoo7TJHidbtS4J5iNmZgXL0FkbFFBjvSfpJIlJ00zbhNYS5f6Guo
 EDmFJl0ZxBHjJnyp378OD8uTs7fLvjx79LjSTbNYiytVbZPQUQ5Yaxu2jXnimvw
 3rrszlaEXAMPLE=
-----END CERTIFICATE-----",
  "keyPair": {
    "PublicKey": "-----BEGIN PUBLIC KEY-----\
MIIBIjANBgkqhkEXAMPLEQEFAAOCAQ8AMIIBCgKCAQEAEXAMPLE1nnyJwKSMHw4h\
MMEXAMPLEuuN/dMAS3fyce8DW/4+EXAMPLEyjmoF/YVF/gHr99VEEXAMPLE5VF13\
59VK7cEXAMPLE67GK+y+jikqXOgHh/xJTwo
+sGpWEXAMPLEDz18xOd2ka4tCzuWEXAMPLEahJbYkCPUBSU8opVkR7qkEXAMPLE1DR6sx2HocliOOLtu6Fkw91swQWE
\\GB3ZPrNh0PzQYvjUStZeccyNCx2EXAMPLEvp9mQOUXP6plfgxwKRX2fEXAMPLEDa\
hJLXkX3rHU2xbxJSq7D+XEXAMPLEcw+LyFhI5mgFRl88eGdsAEXAMPLElnI9EesG\
FQIDAQAB\
-----END PUBLIC KEY-----\
",
    "PrivateKey": "-----BEGIN RSA PRIVATE KEY-----\
key omitted for security reasons\
-----END RSA PRIVATE KEY-----\
"
  }
}
```

Save the certificate's Amazon Resource Name (ARN) to use to configure the certificate later.

Next, configure the thing certificate. For more information, see Configure the thing certificate.

Create a token exchange role

Greengrass core devices use an IAM service role, called the *token exchange role*, to authorize calls to AWS services. The device uses the AWS IoT credentials provider to get temporary AWS credentials for this role, which allows the device to interact with AWS IoT, send logs to Amazon CloudWatch

Logs, and download custom component artifacts from Amazon S3. For more information, see Authorize core devices to interact with AWS services.

You use an AWS IoT *role alias* to configure the token exchange role for Greengrass core devices. Role aliases enable you to change the token exchange role for a device but keep the device configuration the same. For more information, see Authorizing direct calls to AWS services in the *AWS IoT Core Developer Guide*.

In this section, you create a token exchange IAM role and an AWS IoT role alias that points to the role. If you have already set up a Greengrass core device, you can use its token exchange role and role alias instead of creating new ones. Then, you configure your device's AWS IoT thing to use that role and alias.

To create a token exchange IAM role

1. Create an IAM role that your device can use as a token exchange role. Do the following:

   a. Create a file that contains the trust policy document that the token exchange role requires.

      For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

      ```
      nano device-role-trust-policy.json
      ```

      Copy the following JSON into the file.

      ```
      {
        "Version": "2012-10-17",
        "Statement": [
          {
            "Effect": "Allow",
            "Principal": {
              "Service": "credentials.iot.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
          }
        ]
      }
      ```

   b. Create the token exchange role with the trust policy document.

- Replace *GreengrassV2TokenExchangeRole* with the name of the IAM role to create.

```
aws iam create-role --role-name GreengrassV2TokenExchangeRole --assume-role-
policy-document file://device-role-trust-policy.json
```

The response looks similar to the following example, if the request succeeds.

```
{
  "Role": {
    "Path": "/",
    "RoleName": "GreengrassV2TokenExchangeRole",
    "RoleId": "AROAZ2YMUHYHK5OKM77FB",
    "Arn": "arn:aws:iam::123456789012:role/GreengrassV2TokenExchangeRole",
    "CreateDate": "2021-02-06T00:13:29+00:00",
    "AssumeRolePolicyDocument": {
      "Version": "2012-10-17",
      "Statement": [
        {
          "Effect": "Allow",
          "Principal": {
            "Service": "credentials.iot.amazonaws.com"
          },
          "Action": "sts:AssumeRole"
        }
      ]
    }
  }
```

c. Create a file that contains the access policy document that the token exchange role requires.

For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

```
nano device-role-access-policy.json
```

Copy the following JSON into the file.

```
{
  "Version": "2012-10-17",
```

```
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:DescribeLogStreams",
        "s3:GetBucketLocation"
      ],
      "Resource": "*"
    }
  ]
}
```

> **Note**
>
> This access policy doesn't allow access to component artifacts in S3 buckets. To deploy custom components that define artifacts in Amazon S3, you must add permissions to the role to allow your core device to retrieve component artifacts. For more information, see Allow access to S3 buckets for component artifacts. If you don't yet have an S3 bucket for component artifacts, you can add these permissions later after you create a bucket.

d. Create the IAM policy from the policy document.

- Replace *GreengrassV2TokenExchangeRoleAccess* with the name of the IAM policy to create.

```
aws iam create-policy --policy-name GreengrassV2TokenExchangeRoleAccess --
policy-document file://device-role-access-policy.json
```

The response looks similar to the following example, if the request succeeds.

```
{
  "Policy": {
    "PolicyName": "GreengrassV2TokenExchangeRoleAccess",
    "PolicyId": "ANPAZ2YMUHYHACI7C5Z66",
    "Arn": "arn:aws:iam::123456789012:policy/
GreengrassV2TokenExchangeRoleAccess",
```

```
    "Path": "/",
    "DefaultVersionId": "v1",
    "AttachmentCount": 0,
    "PermissionsBoundaryUsageCount": 0,
    "IsAttachable": true,
    "CreateDate": "2021-02-06T00:37:17+00:00",
    "UpdateDate": "2021-02-06T00:37:17+00:00"
  }
}
```

e. Attach the IAM policy to the token exchange role.

- Replace *GreengrassV2TokenExchangeRole* with the name of the IAM role.
- Replace the policy ARN with the ARN of the IAM policy that you created in the previous step.

```
aws iam attach-role-policy --role-name GreengrassV2TokenExchangeRole --policy-
arn arn:aws:iam::123456789012:policy/GreengrassV2TokenExchangeRoleAccess
```

The command doesn't have any output if the request succeeds.

2. Create an AWS IoT role alias that points to the token exchange role.

- Replace *GreengrassCoreTokenExchangeRoleAlias* with the name of the role alias to create.
- Replace the role ARN with the ARN of the IAM role that you created in the previous step.

```
aws iot create-role-alias --role-alias GreengrassCoreTokenExchangeRoleAlias --role-
arn arn:aws:iam::123456789012:role/GreengrassV2TokenExchangeRole
```

The response looks similar to the following example, if the request succeeds.

```
{
  "roleAlias": "GreengrassCoreTokenExchangeRoleAlias",
  "roleAliasArn": "arn:aws:iot:us-west-2:123456789012:rolealias/
GreengrassCoreTokenExchangeRoleAlias"
}
```

> **Note**
>
> To create a role alias, you must have permission to pass the token exchange IAM role to AWS IoT. If you receive an error message when you try to create a role alias, check that your AWS user has this permission. For more information, see Granting a user permissions to pass a role to an AWS service in the *AWS Identity and Access Management User Guide*.

3. Create and attach an AWS IoT policy that allows your Greengrass core device to use the role alias to assume the token exchange role. If you have set up a Greengrass core device before, you can attach its role alias AWS IoT policy instead of creating a new one. Do the following:

   a. (Optional) Create a file that contains the AWS IoT policy document that the role alias requires.

      For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

      ```
      nano greengrass-v2-iot-role-alias-policy.json
      ```

      Copy the following JSON into the file.

      - Replace the resource ARN with the ARN of your role alias.

      ```
      {
        "Version":"2012-10-17",
        "Statement": [
          {
            "Effect": "Allow",
            "Action": "iot:AssumeRoleWithCertificate",
            "Resource": "arn:aws:iot:us-west-2:123456789012:rolealias/
      GreengrassCoreTokenExchangeRoleAlias"
          }
        ]
      }
      ```

   b. Create an AWS IoT policy from the policy document.

- Replace *GreengrassCoreTokenExchangeRoleAliasPolicy* with the name of the AWS IoT policy to create.

```
aws iot create-policy --policy-name GreengrassCoreTokenExchangeRoleAliasPolicy
 --policy-document file://greengrass-v2-iot-role-alias-policy.json
```

The response looks similar to the following example, if the request succeeds.

```
{
  "policyName": "GreengrassCoreTokenExchangeRoleAliasPolicy",
  "policyArn": "arn:aws:iot:us-west-2:123456789012:policy/
GreengrassCoreTokenExchangeRoleAliasPolicy",
  "policyDocument": "{
    \\"Version\\":\\"2012-10-17       \\",
    \\"Statement\\": [
      {
        \\"Effect\\": \\"Allow\\",
        \\"Action\\": \\"iot:AssumeRoleWithCertificate\\",
        \\"Resource\\": \\"arn:aws:iot:us-west-2:123456789012:rolealias/
GreengrassCoreTokenExchangeRoleAlias\\"
      }
    ]
  }",
  "policyVersionId": "1"
}
```

c. Attach the AWS IoT policy to the AWS IoT thing's certificate.

- Replace *GreengrassCoreTokenExchangeRoleAliasPolicy* with the name of the role alias AWS IoT policy.
- Replace the target ARN with the ARN of the certificate for your AWS IoT thing.

```
aws iot attach-policy --policy-name GreengrassCoreTokenExchangeRoleAliasPolicy
 --target arn:aws:iot:us-west-2:123456789012:cert/
aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4
```

The command doesn't have any output if the request succeeds.

Download certificates to the device

Earlier, you downloaded your device's certificate to your development computer. In this section, you download the Amazon root certificate authority (CA) certificate. Then, if you plan to run the AWS IoT Greengrass Core software in Docker on a different computer than your development computer, you copy the certificates to that host computer. The AWS IoT Greengrass Core software uses these certificates to connect to the AWS IoT cloud service.

To download certificates to the device

1. On your development computer, download the Amazon root certificate authority (CA) certificate. AWS IoT certificates are associated with Amazon's root CA certificate by default.

   Linux or Unix

   ```
   sudo curl -o ./greengrass-v2-certs/AmazonRootCA1.pem https://
   www.amazontrust.com/repository/AmazonRootCA1.pem
   ```

   Windows Command Prompt (CMD)

   ```
   curl -o .\greengrass-v2-certs\AmazonRootCA1.pem https://www.amazontrust.com/
   repository/AmazonRootCA1.pem
   ```

   PowerShell

   ```
   iwr -Uri https://www.amazontrust.com/repository/AmazonRootCA1.pem -OutFile .
   \greengrass-v2-certs\AmazonRootCA1.pem
   ```

2. If you plan to run the AWS IoT Greengrass Core software in Docker on a different device than your development computer, copy the certificates to the host computer. If SSH and SCP are enabled on the development computer and the host computer, you can use the `scp` command on your development computer to transfer the certificates. Replace *`device-ip-address`* with the IP address of your host computer.

   ```
   scp -r greengrass-v2-certs/ device-ip-address:~
   ```

Create a configuration file

1. On the host computer, create a folder where you place your configuration file.

```
mkdir ./greengrass-v2-config
```

2. Use a text editor to create a configuration file named `config.yaml` in the `./greengrass-v2-config` folder.

   For example, you can run the following command to use GNU nano to create the `config.yaml`.

   ```
   nano ./greengrass-v2-config/config.yaml
   ```

3. Copy the following YAML content into the file. This partial configuration file specifies system parameters and Greengrass nucleus parameters.

   ```
   ---
   system:
     certificateFilePath: "/tmp/certs/device.pem.crt"
     privateKeyPath: "/tmp/certs/private.pem.key"
     rootCaPath: "/tmp/certs/AmazonRootCA1.pem"
     rootpath: "/greengrass/v2"
     thingName: "MyGreengrassCore"
   services:
     aws.greengrass.Nucleus:
       componentType: "NUCLEUS"
       version: "nucleus-version"
       configuration:
         awsRegion: "region"
         iotRoleAlias: "GreengrassCoreTokenExchangeRoleAlias"
         iotDataEndpoint: "device-data-prefix-ats.iot.region.amazonaws.com"
         iotCredEndpoint: "device-credentials-prefix.credentials.region.amazonaws.com"
   ```

   Then, replace the following values:

   - */tmp/certs*. The directory in the Docker container to which you mount the downloaded certificates when you start the container.
   - */greengrass/v2*. The Greengrass root folder that you want to use for installation. You use the GGC_ROOT environment variable to set this value.
   - *MyGreengrassCore*. The name of the AWS IoT thing.
   - *nucleus-version*. The version of the AWS IoT Greengrass Core software to install. This value must match the version of the Docker image or Dockerfile that you downloaded. If

you downloaded the Greengrass Docker image with the `latest` tag, use **`docker inspect`** ***`image-id`*** to see the image version.

- *`region`*. The AWS Region where you created your AWS IoT resources. You must also specify the same value for the `AWS_REGION` environment variable in your environment file.
- *`GreengrassCoreTokenExchangeRoleAlias`*. The token exchange role alias.
- *`device-data-prefix`*. The prefix for your AWS IoT data endpoint.
- *`device-credentials-prefix`*. The prefix for your AWS IoT credentials endpoint.

Create an environment file

This tutorial uses an environment file to set the environment variables that will be passed to the AWS IoT Greengrass Core software installer inside the Docker container. You can also use the `-e` or `--env` argument in your `docker run` command to set environment variables in the Docker container or you can set the variables in an `environment` block in the `docker-compose.yml` file.

1. Use a text editor to create an environment file named `.env`.

   For example, on a Linux-based system, you can run the following command to use GNU nano to create the `.env` in the current directory.

   ```
   nano .env
   ```

2. Copy the following content into the file.

   ```
   GGC_ROOT_PATH=/greengrass/v2
   AWS_REGION=region
   PROVISION=false
   COMPONENT_DEFAULT_USER=ggc_user:ggc_group
   INIT_CONFIG=/tmp/config/config.yaml
   ```

   Then, replace the following values.

   - */greengrass/v2*. The path to the root folder to use to install the AWS IoT Greengrass Core software.
   - *`region`*. The AWS Region where you created your AWS IoT resources. You must specify the same value for the `awsRegion` configuration parameter in your configuration file.

- */tmp/config/*. The folder where you mount the configuration file when you start the Docker container.

> **Note**
>
> You can set the `DEPLOY_DEV_TOOLS` environment variable to `true` to deploy the Greengrass CLI component, which enables you to develop custom components inside of the Docker container. We recommend that you use this component in only development environments, not production environments. This component provides access to information and operations that you typically won't need in a production environment. Follow the principle of least privilege by deploying this component to only core devices where you need it.

Run the AWS IoT Greengrass Core software in a container

This tutorial shows you how to start the Docker image that you built in a Docker container. You can use the Docker CLI or the Docker Compose CLI to run the AWS IoT Greengrass Core software image in a Docker container.

Docker

- This tutorial shows you how to start the Docker image that you built in a the Docker container.

  ```
  docker run --rm --init -it --name docker-image \
   -v path/to/greengrass-v2-config:/tmp/config/:ro \
   -v path/to/greengrass-v2-certs:/tmp/certs:ro \
   --env-file .env \
   -p 8883 \
   your-container-image:version
  ```

  This example command uses the following arguments for docker run:

  - `--rm`. Cleans up the container when it exits.
  - `--init`. Uses an init process in the container.

> **Note**
>
> The `--init` argument is required to shut down AWS IoT Greengrass Core software when you stop the Docker container.

- `-it`. (Optional) Runs the Docker container in the foreground as an interactive process. You can replace this with the `-d` argument to run the Docker container in detached mode instead. For more information, see Detached vs foreground in the Docker documentation.
- `--name`. Runs a container named `aws-iot-greengrass`
- `-v`. Mounts a volume into the Docker container to make the configuration file and the certificate files available to AWS IoT Greengrass running inside the container.
- `--env-file`. (Optional) Specifies the environment file to set the environment variables that will be passed to the AWS IoT Greengrass Core software installer inside the Docker container. This argument is required only if you created an environment file to set environment variables. If you didn't create an environment file, you can use `--env` arguments to set environment variables directly in your Docker run command.
- `-p`. (Optional) Publishes the 8883 container port to the host machine. This argument is required if you want to connect and communicate over MQTT because AWS IoT Greengrass uses port 8883 for MQTT traffic. To open other ports, use additional `-p` arguments.

> **Note**
>
> To run your Docker container with increased security, you can use the `--cap-drop` and `--cap-add` arguments to selectively enable Linux capabilities for your container. For more information, see Runtime privilege and Linux capabilities in the Docker documentation.

Docker Compose

1. Use a text editor to create a Docker Compose file named `docker-compose.yml`.

   For example, on a Linux-based system, you can run the following command to use GNU nano to create the `docker-compose.yml` in the current directory.

```
nano docker-compose.yml
```

> **Note**
>
> You can also download and use the latest version of the AWS-provided Compose file from GitHub.

2. Add the following content to the Compose file. Your file should look similar to the following example. Replace *your-container-name:version* with the name of your Docker image.

```
version: '3.7'

services:
  greengrass:
    init: true
    build:
      context: .
    container_name: aws-iot-greengrass
    image: your-container-name:version
    volumes:
      - /path/to/greengrass-v2-config:/tmp/config/:ro
      - /path/to/greengrass-v2-certs:/tmp/certs:ro
    env_file: .env
    ports:
      - "8883:8883"
```

The following parameters in this example Compose file are optional:

- `ports`—Publishes the 8883 container ports to the host machine. This parameter is required if you want to connect and communicate over MQTT because AWS IoT Greengrass uses port 8883 for MQTT traffic.
- `env_file`—Specifies the environment file to set the environment variables that will be passed to the AWS IoT Greengrass Core software installer inside the Docker container. This parameter is required only if you created an environment file to set environment variables. If you didn't create an environment file, you can use the `environment` parameter to set the variables directly in your Compose file.

> **Note**
>
> To run your Docker container with increased security, you can use `cap_drop` and `cap_add` in your Compose file to selectively enable Linux capabilities for your container. For more information, see Runtime privilege and Linux capabilities in the Docker documentation.

3. Run the following command to start the container.

```
docker-compose -f docker-compose.yml up
```

Next steps

AWS IoT Greengrass Core software is now running in a Docker container. Run the following command to retrieve the container ID for the currently running container.

```
docker ps
```

You can then run the following command to access the container and explore AWS IoT Greengrass Core software running inside the container.

```
docker exec -it container-id /bin/bash
```

For information about creating a simple component, see Step 4: Develop and test a component on your device in Tutorial: Getting started with AWS IoT Greengrass V2

> **Note**
>
> When you use `docker exec` to run commands inside the Docker container, those commands are not logged in the Docker logs. To log your commands in the Docker logs, attach an interactive shell to the Docker container. For more information, see Attach an interactive shell to the Docker container.

The AWS IoT Greengrass Core log file is called `greengrass.log` and is located in */greengrass/v2*`/logs`. Component log files are also located in the same directory. To copy Greengrass logs to a temporary directory on the host, run the following command:

```
docker cp container-id:/greengrass/v2/logs /tmp/logs
```

If you want to persist logs after a container exits or has been removed, we recommend that you bind-mount only the */greengrass/v2*`/logs` directory to the temporary logs directory on the host instead of mounting the entire Greengrass directory. For more information, see Persist Greengrass logs outside of the Docker container.

To stop a running AWS IoT Greengrass Docker container, run `docker stop` or `docker-compose -f docker-compose.yml stop`. This action sends `SIGTERM` to the Greengrass process and shuts down all associated processes that were started in the container. The Docker container is initialized with the `docker-init` executable as process PID 1, which helps in removing any leftover zombie processes. For more information, see the Specify an init process in the Docker documentation.

For information about troubleshooting issues with running AWS IoT Greengrass in a Docker container, see Troubleshooting AWS IoT Greengrass in a Docker container.

Troubleshooting AWS IoT Greengrass in a Docker container

Use the following information to help you troubleshoot issues with running AWS IoT Greengrass in a Docker container and to debug issues with AWS IoT Greengrass in the Docker container.

Topics

- Troubleshooting issues with running the Docker container
- Debugging AWS IoT Greengrass in a Docker container

Troubleshooting issues with running the Docker container

Use the following information to help troubleshoot issues with running AWS IoT Greengrass in a Docker container.

Topics

- Error: Cannot perform an interactive login from a non TTY device
- Error: Unknown options: -no-include-email
- Error: A firewall is blocking file Sharing between windows and the containers.

- Error: An error occurred (AccessDeniedException) when calling the GetAuthorizationToken operation: User: arn:aws:iam::account-id:user/<user-name> is not authorized to perform: ecr:GetAuthorizationToken on resource: *
- Error: You have reached your pull rate limit

Error: Cannot perform an interactive login from a non TTY device

This error can occur when you run the `aws ecr get-login-password` command. Make sure that you installed the latest AWS CLI version 2 or version 1. We recommend that you use the AWS CLI version 2. For more information, see Installing the AWS CLI in the *AWS Command Line Interface User Guide*.

Error: Unknown options: -no-include-email

This error can occur when you run the `aws ecr get-login` command. Make sure that you have the latest AWS CLI version installed (for example, Run: `pip install awscli --upgrade --user`). For more information, see Installing the AWS Command Line Interface on Microsoft Windows in the *AWS Command Line Interface User Guide*.

Error: A firewall is blocking file Sharing between windows and the containers.

You might receive this error or a `Firewall Detected` message when running Docker on a Windows computer. This can also occur if you are signed in on a virtual private network (VPN) and your network settings are preventing the shared drive from being mounted. In that situation, turn off VPN and re-run the Docker container.

Error: An error occurred (AccessDeniedException) when calling the GetAuthorizationToken operation: User: arn:aws:iam::*account-id*:user/<user-name> is not authorized to perform: ecr:GetAuthorizationToken on resource: *

You might receive this error when running the `aws ecr get-login-password` command if you don't have sufficient permissions to access an Amazon ECR repository. For more information, see Amazon ECR Repository Policy Examples and Accessing One Amazon ECR Repository in the *Amazon ECR User Guide*.

Error: You have reached your pull rate limit

Docker Hub limits the number of pull requests that anonymous and Free Docker Hub users can make. If you exceed the rate limits for anonymous or free user pull requests, then you receive one of the following errors:

```
ERROR: toomanyrequests: Too Many Requests.
```

```
You have reached your pull rate limit.
```

To resolve these errors, you can wait for a few hours before you try another pull request. If you plan on consistently submitting a large number of pull requests, see the Docker Hub website for information about rate limits, and options for authenticating and upgrading your Docker account.

Debugging AWS IoT Greengrass in a Docker container

To debug issues with a Docker container, you can persist the Greengrass runtime logs or attach an interactive shell to the Docker container.

Persist Greengrass logs outside of the Docker container

After you stop a AWS IoT Greengrass container, you can use the following `docker cp` command to copy the Greengrass logs from the Docker container to a temporary logs directory.

```
docker cp container-id:/greengrass/v2/logs /tmp/logs
```

To persist logs even after a container exits or is removed, you must run the AWS IoT Greengrass Docker container after bind-mounting the */greengrass/v2*/logs directory.

To bind-mount the */greengrass/v2*/logs directory, do one of the following when you run a new AWS IoT Greengrass Docker container.

- Include `-v /tmp/logs:/greengrass/v2/logs:ro` in your `docker run` command.

  Modify the `volumes` block in the Compose file to include the following line before you run your `docker-compose up` command.

  ```
  volumes:
   - /tmp/logs:/greengrass/v2/logs:ro
  ```

You can then check your logs at */tmp/logs* on your host to see Greengrass logs while AWS IoT Greengrass is running inside the Docker container.

For information about running Greengrass Docker containers, see Run AWS IoT Greengrass in Docker with manual provisioning and Run AWS IoT Greengrass in Docker with automatic provisioning

Attach an interactive shell to the Docker container

When you use `docker exec` to run commands inside the Docker container, those commands are not captured in the Docker logs. Logging your commands in the Docker logs can help you investigate the state of the Greengrass Docker container. Do one of the following:

- Run the following command in a separate terminal to attach your terminal's standard input, output, and error to the running container. This enables you to view and control the Docker container from your current terminal.

  ```
  docker attach container-id
  ```

- Run the following command in a separate terminal. This enables you to run your commands in interactive mode, even if the container is not attached.

  ```
  docker exec -it container-id sh -c "command > /proc/1/fd/1"
  ```

For general AWS IoT Greengrass troubleshooting, see *Troubleshooting*.

Configure the AWS IoT Greengrass Core software

The AWS IoT Greengrass Core software provides options that you can use to configure the software. You can create deployments to configure the AWS IoT Greengrass Core software on each core device.

Topics

- Deploy the Greengrass nucleus component
- Configure the Greengrass nucleus as a system service
- Control memory allocation with JVM options
- Configure the user that runs components
- Configure system resource limits for components
- Connect on port 443 or through a network proxy

- Use a device certificate signed by a private CA
- Configure MQTT timeouts and cache settings
- Configure Greengrass Nucleus on IPv6 network

Deploy the Greengrass nucleus component

AWS IoT Greengrass provides the AWS IoT Greengrass Core software as a component that you can deploy to your Greengrass core devices. You can create a deployment to apply the same configuration to multiple Greengrass core devices. For more information, see Greengrass nucleus and Update the AWS IoT Greengrass Core software (OTA).

Configure the Greengrass nucleus as a system service

You must configure the AWS IoT Greengrass Core software as a system service in your device's init system to do the following:

- Start the AWS IoT Greengrass Core software when the device boots. This is a good practice if you manage large fleets of devices.
- Install and run plugin components. Several AWS-provided components are plugin components, which enables them to interface directly with the Greengrass nucleus. For more information about component types, see Component types.
- Apply over-the-air (OTA) updates to the core device's AWS IoT Greengrass Core software. For more information, see Update the AWS IoT Greengrass Core software (OTA).
- Enable components to restart the AWS IoT Greengrass Core software or the core device when a deployment updates the component to a new version or updates certain configuration parameters. For more information, see the bootstrap lifecycle step.

> **Important**
>
> On Windows core devices, you must set up the AWS IoT Greengrass Core software as a system service.

Topics

- Configure the nucleus as a system service (Linux)

- Configure the nucleus as a system service (Windows)

Configure the nucleus as a system service (Linux)

Linux devices support different init systems, such as initd, systemd, and SystemV. You use the `--setup-system-service true` argument when you install the AWS IoT Greengrass Core software to start the nucleus as a system service and configure it to launch when the device boots. The installer configures the AWS IoT Greengrass Core software as a system service with systemd.

You can also manually configure the nucleus to run as a system service. The following example is a service file for systemd.

```
[Unit]
Description=Greengrass Core

[Service]
Type=simple
PIDFile=/greengrass/v2/alts/loader.pid
RemainAfterExit=no
Restart=on-failure
RestartSec=10
ExecStart=/bin/sh /greengrass/v2/alts/current/distro/bin/loader

[Install]
WantedBy=multi-user.target
```

After you configure the system service, you can run the following commands to configure starting the device on boot and to start or stop the AWS IoT Greengrass Core software.

- To check the status of the service (systemd)

  ```
  sudo systemctl status greengrass.service
  ```

- To enable the nucleus to start when the device boots.

  ```
  sudo systemctl enable greengrass.service
  ```

- To stop the nucleus from starting when the device boots.

  ```
  sudo systemctl disable greengrass.service
  ```

- To start the AWS IoT Greengrass Core software.

```
sudo systemctl start greengrass.service
```

- To stop the AWS IoT Greengrass Core software.

```
sudo systemctl stop greengrass.service
```

Configure the nucleus as a system service (Windows)

You use the `--setup-system-service true` argument when you install the AWS IoT Greengrass Core software to start the nucleus as a Windows service and configure it to launch when the device boots.

After you configure the service, you can run the following commands to configure starting the device on boot and to start or stop the AWS IoT Greengrass Core software. You must run Command Prompt or PowerShell as an administrator to run these commands.

Windows Command Prompt (CMD)

- To check the status of the service

```
sc query "greengrass"
```

- To enable the nucleus to start when the device boots.

```
sc config "greengrass" start=auto
```

- To stop the nucleus from starting when the device boots.

```
sc config "greengrass" start=disabled
```

- To start the AWS IoT Greengrass Core software.

```
sc start "greengrass"
```

- To stop the AWS IoT Greengrass Core software.

```
sc stop "greengrass"
```

> **Note**
>
> On Windows devices, the AWS IoT Greengrass Core software ignores this shutdown signal while it shuts down Greengrass component processes. If the AWS IoT Greengrass Core software ignores the shutdown signal when you run this command, wait a few seconds, and try again.

PowerShell

- To check the status of the service

```
Get-Service -Name "greengrass"
```

- To enable the nucleus to start when the device boots.

```
Set-Service -Name "greengrass" -Status stopped -StartupType automatic
```

- To stop the nucleus from starting when the device boots.

```
Set-Service -Name "greengrass" -Status stopped -StartupType disabled
```

- To start the AWS IoT Greengrass Core software.

```
Start-Service -Name "greengrass"
```

- To stop the AWS IoT Greengrass Core software.

```
Stop-Service -Name "greengrass"
```

> **Note**
>
> On Windows devices, the AWS IoT Greengrass Core software ignores this shutdown signal while it shuts down Greengrass component processes. If the AWS IoT Greengrass Core software ignores the shutdown signal when you run this command, wait a few seconds, and try again.

Control memory allocation with JVM options

If you're running AWS IoT Greengrass on a device with limited memory, you can use Java virtual machine (JVM) options to control the maximum heap size, garbage collection modes, and compiler options, which control the amount of memory that AWS IoT Greengrass Core software uses. The heap size in the JVM determines how much memory an application can use before garbage collection occurs, or before the application runs out of memory. The maximum heap size specifies the maximum amount of memory the JVM can allocate when expanding the heap during heavy activity.

To control memory allocation, create a new deployment or revise an existing deployment that includes the nucleus component, and specify your JVM options in the `jvmOptions` configuration parameter in the nucleus component configuration.

Depending on your requirements, you can run AWS IoT Greengrass Core software with reduced memory allocation or with minimum memory allocation.

Reduced memory allocation

To run AWS IoT Greengrass Core software with reduced memory allocation, we recommend that you use the following example configuration merge update to set JVM options in your nucleus configuration:

```
{
  "jvmOptions": "-XX:+UseSerialGC -XX:TieredStopAtLevel=1"
}
```

Minimum memory allocation

To run AWS IoT Greengrass Core software with minimum memory allocation, we recommend that you use the following example configuration merge update to set JVM options in your nucleus configuration:

```
{
  "jvmOptions": "-Xmx32m -XX:+UseSerialGC -Xint"
}
```

> **Important**
>
> Running AWS IoT Greengrass Core software with minimum memory allocation can have a significant performance impact on low spec systems because the JVM will do more processing when using less memory. We recommend tuning the options to balance your memory and performance needs.

These example configuration merge updates use the following JVM options:

`-XX:+UseSerialGC`

Specifies to use serial garbage collection for JVM heap space. The serial garbage collector is slower, but uses less memory than other JVM garbage collection implementations.

`-XX:TieredStopAtLevel=1`

Instructs the JVM to use the Java just-in-time (JIT) compiler once. Because JIT compiled code uses space in the device memory, using the JIT compiler more than once consumes more memory than a single compilation.

`-Xmx`*NN*`m`

Sets the maximum JVM heap size.

> **Important**
>
> Setting the maximum heap size too low can cause slower performance or out-of-memory errors. We recommend measuring your current heap usage before setting a maximum size with the `-Xmx`*NN*`m` option. Configure your JVM with the `-XX:NativeMemoryTracking=detail` JVM option. Then, measure your current heap usage by using the `VM.native_memory` command request within the jcmd Utility.

If measurement of the heap is not an option, use `-Xmx64m` as a starting value to limit the heap size to 64 MB. You can then incrementally decrease the max heap size from there. For minimum memory allocation, use `-Xmx32m` as a starting value to limit the heap size to 32 MB.

You can increase or decrease the `-Xmx` value depending on your actual requirements; however, we strongly recommend that you don't set the maximum heap size below 16 MB. The amount

of JVM heap size needed can also vary over time based on the plugin components deployed to the core device. If the maximum heap size is too low for your environment, then the AWS IoT Greengrass Core software might encounter unexpected errors because of insufficient memory. If you experience a slower performance or encounter errors because of insufficient memory, revert to a known good setting. For example, if your normal committed heap size is `41428KB`, use `-Xmx40m` to slightly limit heap usage.

`-Xint`

Instructs the JVM not to use the just-in-time (JIT) compiler. Instead, the JVM runs in interpreted-only mode. This mode is slower (potentially 20 times slower for deployments on low-end systems) than running JIT compiled code; however, the compiled code doesn't use any space in memory.

For information about creating configuration merge updates, see Update component configurations.

Configure the user that runs components

The AWS IoT Greengrass Core software can run component processes as a system user and group different from the one that runs the software. This increases security, because you can run the AWS IoT Greengrass Core software as root, or as an administrator user, without giving those permissions to components that run on the core device.

The following table indicates which types of components the AWS IoT Greengrass Core software can run as a user that you specify. For more information, see Component types.

Component type	Configure component user
Nucleus	⊗ No
Plugin	⊗ No
Generic	⊘ Yes
Lambda (non-containerized)	⊘ Yes
Lambda (containerized)	⊘ Yes

You must create the component user before you can specify it in a deployment configuration. On Windows-based devices, you must also store the user name and password for the user in the credential manager instance of the LocalSystem account. For more information, see Set up a component user on Windows devices.

When you configure the component user on a Linux-based device, you can optionally also specify a group. You specify the user and group separated by a colon (`:`) in the following format: *`user`*`:`*`group`*. If you don't specify a group, the AWS IoT Greengrass Core software defaults to the primary group of the user. You can use either the name or the ID to identify the user and group.

On Linux-based devices, you can also run components as a system user that doesn't exist, also called an unknown user, to increase security. A Linux process can signal any other process that is run by the same user. An unknown user doesn't run other processes, so you can run components as an unknown user to prevent components from signaling other components on the core device. To run components as an unknown user, specify a user ID that doesn't exist on the core device. You can also specify a group ID that doesn't exist to run as an unknown group.

You can configure the user for each component and for each core device.

- **Configure for a component**

  You can configure each component to run with a user specific to that component. When you create a deployment, you can specify the user for each component in the `runWith` configuration for that component. The AWS IoT Greengrass Core software runs components as the specified user if you configure them. Otherwise, it defaults to run components as the default user that you configure for the core device. For more information about specifying the component user in the deployment configuration, see the `runWith` configuration parameter in Create deployments.

- **Configure default user for a core device**

  You can configure a default user that the AWS IoT Greengrass Core software uses to run components. When the AWS IoT Greengrass Core software runs a component, it checks if you specified a user for that component, and uses it to run the component. If the component doesn't specify a user, then the AWS IoT Greengrass Core software runs the component as the default user that you configured for the core device. For more information, see Configure the default component user.

> **Note**
>
> On Windows-based devices, you must specify at least a default user to run components.
>
> On Linux-based devices, the following considerations apply if you don't configure a user to run components:
>
> - If you run the AWS IoT Greengrass Core software as root, then the software won't run components. You must specify a default user to run components if you run as root.
> - If you run the AWS IoT Greengrass Core software as a non-root user, then the software runs components as that user.

Topics

- Set up a component user on Windows devices
- Configure the default component user

Set up a component user on Windows devices

To set up a component user on a Windows-based device

1. Create the component user in the LocalSystem account on the device.

   ```
   net user /add component-user password
   ```

2. Use Microsoft's PsExec utility to store the user name and password for the component user in the Credential Manager instance for the LocalSystem account.

   ```
   psexec -s cmd /c cmdkey /generic:component-user /user:component-user /pass:password
   ```

   > **Note**
   >
   > On Windows-based devices, the LocalSystem account runs the Greengrass nucleus, and you must use the PsExec utility to store the component user information in the LocalSystem account. Using the Credential Manager application stores this information in the Windows account of the currently logged on user, instead of the LocalSystem account.

Configure the default component user

You can use a deployment to configure the default user on a core device. In this deployment, you update the nucleus component configuration.

> **Note**
>
> You can also set the default user when you install the AWS IoT Greengrass Core software with the `--component-default-user` option. For more information, see Install the AWS IoT Greengrass Core software.

Create a deployment that specifies the following configuration update for the `aws.greengrass.Nucleus` component.

Linux

```
{
  "runWithDefault": {
    "posixUser": "ggc_user:ggc_group"
  }
}
```

Windows

```
{
  "runWithDefault": {
    "windowsUser": "ggc_user"
  }
}
```

> **Note**
>
> The user that you specify must exist, and the user name and password for this user must be stored in the credential manager instance of the LocalSystem account on your Windows device. For more information, see Set up a component user on Windows devices.

The following example defines a deployment for a Linux-based device that configures `ggc_user` as the default user and `ggc_group` as the default group. The `merge` configuration update requires a serialized JSON object.

```
{
  "components": {
    "aws.greengrass.Nucleus": {
      "version": "2.18.4",
      "configurationUpdate": {
        "merge": "{\"runWithDefault\":{\"posixUser\":\"ggc_user:ggc_group\"}}"
      }
    }
  }
}
```

Configure system resource limits for components

> **Note**
>
> This feature is available for v2.4.0 and later of the Greengrass nucleus component. AWS IoT Greengrass doesn't currently support this feature on Windows core devices.

You can configure the maximum amount of CPU and RAM usage that each component's processes can use on the core device.

The following table shows the types of components that support system resource limits. For more information, see Component types.

Component type	Configure system resource limits
Nucleus	⊗ No
Plugin	⊗ No
Generic	⊘ Yes
Lambda (non-containerized)	⊘ Yes
Lambda (containerized)	⊗ No

> **Important**
>
> System resource limits aren't supported when you run AWS IoT Greengrass Core software in a Docker container.

You can configure system resource limits for each component and for each core device.

- **Configure for a component**

  You can configure each component with system resource limits specific to that component. When you create a deployment, you can specify the system resource limits for each component in the deployment. If the component supports system resource limits, the AWS IoT Greengrass Core software applies the limits to the component's processes. If you don't specify system resource limits for a component, the AWS IoT Greengrass Core software uses any defaults that you have configured for the core device. For more information, see Create deployments.

- **Configure defaults for a core device**

  You can configure the default system resource limits that the AWS IoT Greengrass Core software applies to components that support these limits. When the AWS IoT Greengrass Core software runs a component, it applies the system resource limits that you specify for that component. If that component doesn't specify system resource limits, the the AWS IoT Greengrass Core software applies the default system resource limits that you configure for the core device. If you don't specify default system resource limits, the AWS IoT Greengrass Core software doesn't apply any system resource limits by default. For more information, see Configure default system resource limits.

Configure default system resource limits

You can deploy the Greengrass nucleus component to configure the default system resource limits for a core device. To configure the default system resource limits, create a deployment that specifies the following configuration update for the `aws.greengrass.Nucleus` component.

```
{
  "runWithDefault": {
    "systemResourceLimits": {
      "cpu": cpuTimeLimit,
      "memory": memoryLimitInKb
```

```
    }
  }
}
```

The following example defines a deployment that configures the CPU time limit to 2, which is equivalent to 50% usage on a device with 4 CPU cores. This example also configures the memory usage to 100 MB.

```
{
  "components": {
    "aws.greengrass.Nucleus": {
      "version": "2.18.4",
      "configurationUpdate": {
        "merge": "{\"runWithDefault\":{\"systemResourceLimits\":\"cpus\":2,\"memory
\":102400}}}"
      }
    }
  }
}
```

Connect on port 443 or through a network proxy

AWS IoT Greengrass core devices communicate with AWS IoT Core using the MQTT messaging protocol with TLS client authentication. By convention, MQTT over TLS uses port 8883. However, as a security measure, restrictive environments might limit inbound and outbound traffic to a small range of TCP ports. For example, a corporate firewall might open port 443 for HTTPS traffic, but close other ports that are used for less common protocols, such as port 8883 for MQTT traffic. Other restrictive environments might require all traffic to go through a proxy before connecting to the internet.

> **Note**
>
> Greengrass core devices that run Greengrass nucleus component v2.0.3 and earlier use port 8443 to connect to the AWS IoT Greengrass data plane endpoint. These devices must be able to connect to this endpoint on port 8443. For more information, see Allow device traffic through a proxy or firewall.

To enable communication in these scenarios, AWS IoT Greengrass provides the following configuration options:

- **MQTT communication over port 443**. If your network allows connections to port 443, you can configure the Greengrass core device to use port 443 for MQTT traffic instead of the default port 8883. This can be a direct connection to port 443 or a connection through a network proxy server. Unlike the default configuration, which uses certificate-based client authentication, MQTT on port 443 uses the device service role for authentication.

  For more information, see Configure MQTT over port 443.
- **HTTPS communication over port 443**. The AWS IoT Greengrass Core software sends HTTPS traffic over port 8443 by default, but you can configure it to use port 443. AWS IoT Greengrass uses the Application Layer Protocol Network (ALPN) TLS extension to enable this connection. As with the default configuration, HTTPS on port 443 uses certificate-based client authentication.

  > **Important**
  >
  > To use ALPN and enable HTTPS communication over port 443, your core device must run Java 8 update 252 or later. All updates of Java version 9 and later also support ALPN.

  For more information, see Configure HTTPS over port 443.
- **Connection through a network proxy**. You can configure a network proxy server to act as an intermediary for connecting to the Greengrass core device. AWS IoT Greengrass supports basic authentication for HTTP and HTTPS proxies.

  Greengrass core devices must run Greengrass nucleus v2.5.0 or later to use HTTPS proxies.

  The AWS IoT Greengrass Core software passes the proxy configuration to components through the `ALL_PROXY`, `HTTP_PROXY`, `HTTPS_PROXY`, and `NO_PROXY` environment variables. Components must use these settings to connect through the proxy. Components use common libraries (such as boto3, cURL, and the python `requests` package) that typically use these environment variables by default to make connections. If a component also specifies these environment variables, AWS IoT Greengrass doesn't override them.

  For more information, see Configure a network proxy.

Configure MQTT over port 443

You can configure MQTT over port 443 on existing core devices or when you install the AWS IoT Greengrass Core software on a new core device.

Topics

- Configure MQTT over port 443 on existing core devices
- Configure MQTT over port 443 during installation

Configure MQTT over port 443 on existing core devices

You can use a deployment to configure MQTT over port 443 on a single core device or a group of core devices. In this deployment, you update the nucleus component configuration. The nucleus restarts when you update its `mqtt` configuration.

To configure MQTT over port 443, create a deployment that specifies the following configuration update for the `aws.greengrass.Nucleus` component.

```
{
  "mqtt": {
    "port": 443
  }
}
```

The following example defines a deployment that configures MQTT over port 443. The `merge` configuration update requires a serialized JSON object.

```
{
  "components": {
    "aws.greengrass.Nucleus": {
      "version": "2.18.4",
      "configurationUpdate": {
        "merge": "{\"mqtt\":{\"port\":443}}"
      }
    }
  }
}
```

Configure MQTT over port 443 during installation

You can configure MQTT over port 443 when you install the AWS IoT Greengrass Core software on a core device. Use the `--init-config` installer argument to configure MQTT over port 443. You can specify this argument when you install with manual provisioning, fleet provisioning, or custom provisioning.

Configure HTTPS over port 443

This feature requires Greengrass nucleus v2.0.4 or later.

You can configure HTTPS over port 443 on existing core devices or when you install the AWS IoT Greengrass Core software on a new core device.

Topics

- Configure HTTPS over port 443 on existing core devices
- Configure HTTPS over port 443 during installation

Configure HTTPS over port 443 on existing core devices

You can use a deployment to configure HTTPS over port 443 on a single core device or a group of core devices. In this deployment, you update the nucleus component configuration.

To configure HTTPS over port 443, create a deployment that specifies the following configuration update for the `aws.greengrass.Nucleus` component.

```
{
  "greengrassDataPlanePort": 443
}
```

The following example defines a deployment that configures HTTPS over port 443. The `merge` configuration update requires a serialized JSON object.

```
{
  "components": {
    "aws.greengrass.Nucleus": {
      "version": "2.18.4",
      "configurationUpdate": {
        "merge": "{\"greengrassDataPlanePort\":443}"
      }
    }
  }
}
```

Configure HTTPS over port 443 during installation

You can configure HTTPS over port 443 when you install the AWS IoT Greengrass Core software on a core device. Use the `--init-config` installer argument to configure HTTPS over port 443. You

can specify this argument when you install with manual provisioning, fleet provisioning, or custom provisioning.

Configure a network proxy

Follow a procedure in this section to configure Greengrass core devices to connect to the internet through an HTTP or HTTPS network proxy. For more information about the endpoints and ports that core devices use, see Allow device traffic through a proxy or firewall.

Important

If your core device runs a version of the Greengrass nucleus earlier than v2.4.0, your device's role must allow the following permissions to use a network proxy:

- `iot:Connect`
- `iot:Publish`
- `iot:Receive`
- `iot:Subscribe`

This is necessary because the device uses AWS credentials from the token exchange service to authenticate MQTT connections to AWS IoT. The device uses MQTT to receive and install deployments from the AWS Cloud, so your device won't work unless you define these permissions on its role. Devices typically use X.509 certificates to authenticate MQTT connections, but devices can't do this to authenticate when they use a proxy.
For more information about how to configure the device role, see Authorize core devices to interact with AWS services.

Topics

- Configure a network proxy on existing core devices
- Configure a network proxy during installation
- Enable the core device to trust an HTTPS proxy
- The networkProxy object

Configure a network proxy on existing core devices

You can use a deployment to configure a network proxy on a single core device or a group of core devices. In this deployment, you update the nucleus component configuration. The nucleus restarts when you update its `networkProxy` configuration.

To configure a network proxy, create a deployment for the `aws.greengrass.Nucleus` component that merges the following configuration update. This configuration update contains the networkProxy object.

```
{
  "networkProxy": {
    "noProxyAddresses": "http://192.168.0.1,www.example.com",
    "proxy": {
      "url": "https://my-proxy-server:1100"
    }
  }
}
```

The following example defines a deployment that configures a network proxy. The `merge` configuration update requires a serialized JSON object.

```
{
  "components": {
    "aws.greengrass.Nucleus": {
      "version": "2.18.4",
      "configurationUpdate": {
        "merge": "{\"networkProxy\":{\"noProxyAddresses\":
\"http://192.168.0.1,www.example.com\",\"proxy\":{\"url\":\"https://my-proxy-
server:1100\",\"username\":\"Mary_Major\",\"password\":\"pass@word1357\"}}}"
      }
    }
  }
}
```

Configure a network proxy during installation

You can configure a network proxy when you install the AWS IoT Greengrass Core software on a core device. Use the `--init-config` installer argument to configure the network proxy. You can specify this argument when you install with manual provisioning, fleet provisioning, or custom provisioning.

Enable the core device to trust an HTTPS proxy

When you configure a core device to use an HTTPS proxy, you must add the proxy server certificate chain to the core device's to enable it to trust the HTTPS proxy. Otherwise, the core device might encounter errors when it tries to route traffic through the proxy. Add the proxy server CA certificate to the core device's Amazon root CA certificate file.

To enable the core device to trust the HTTPS proxy

1. Find the Amazon root CA certificate file on the core device.

   - If you installed the AWS IoT Greengrass Core software with automatic provisioning, the Amazon root CA certificate file exists at */greengrass/v2*/`rootCA.pem`.
   - If you installed the AWS IoT Greengrass Core software with manual or fleet provisioning, the Amazon root CA certificate file might exist at */greengrass/v2*/`AmazonRootCA1.pem`.

   If the Amazon root CA certificate doesn't exist at these locations, check the `system.rootCaPath` property in */greengrass/v2*/`config/effectiveConfig.yaml` to find its location.
2. Add the contents of the proxy server CA certificate file to the Amazon root CA certificate file.

   The following example shows a proxy server CA certificate added to the Amazon root CA certificate file.

```
-----BEGIN CERTIFICATE-----
MIIEFTCCAv2gAwIQWgIVAMHSAzWG/5YVRYtRQOxXUTEpHuEmApzGCSqGSIb3DQEK
\nCwUAhuL9MQswCQwJVUzEPMAVUzEYMBYGA1UECgwP1hem9uLmNvbSBJbmMuMRww
... content of proxy CA certificate ...
+vHIRlt0e5JAm5\noTIZGoFbK82A0/nO7f/t5PSIDAim9V3Gc3pSXxCCAQoFYnui
GaPUlGk1gCE84a0X\n7Rp/lND/PuMZ/s8YjlkY2NmYmNjMCAXDTE5MTEyN2cM216
gJMIADggEPADf2/m45hzEXAMPLE=
-----END CERTIFICATE-----

-----BEGIN CERTIFICATE-----
MIIDQTCCAimgF6AwIBAgITBmyfz/5mjAo54vB4ikPmljZKyjANJmApzyMZFo6qBg
ADA5MQswCQYDVQQGEwJVUzEPMA0tMVT8QtPHRh8jrdkGA1UEChMGDV3QQDExBBKW
... content of root CA certificate ...
o/ufQJQWUCyziar1hem9uMRkwFwYVPSHCb2XV4cdFyQzR1KldZwgJcIQ6XUDgHaa
5MsI+yMRQ+hDaXJiobldXgjUka642M4UwtBV8oK2xJNDd2ZhwLnoQdeXeGADKkpy
rqXRfKoQnoZsG4q5WTP46EXAMPLE
```

```
-----END CERTIFICATE-----
```

The networkProxy object

Use the `networkProxy` object to specify information about the network proxy. This object contains the following information:

`noProxyAddresses`

(Optional) A comma-separated list of IP addresses or host names that are exempt from the proxy.

`proxy`

The proxy to which to connect. This object contains the following information:

`url`

The URL of the proxy server in the format `scheme://userinfo@host:port`.

- `scheme` – The scheme, which must be `http` or `https`.

  Important

  Greengrass core devices must run Greengrass nucleus v2.5.0 or later to use HTTPS proxies.
  If you configure an HTTPS proxy, you must add the proxy server CA certificate to the core device's Amazon root CA certificate. For more information, see Enable the core device to trust an HTTPS proxy.

- `userinfo` – (Optional) The user name and password information. If you specify this information in the `url`, the Greengrass core device ignores the `username` and `password` fields.
- `host` – The host name or IP address of the proxy server.
- `port` – (Optional) The port number. If you don't specify the port, then the Greengrass core device uses the following default values:
  - `http` – 80
  - `https` – 443

`username`

(Optional) The user name that authenticates the proxy server.

`password`

(Optional) The password that authenticates the proxy server.

Use a device certificate signed by a private CA

If you are using a custom private certificate authority (CA), you must set the Greengrass nucleus' **`greengrassDataPlaneEndpoint`** to **`iotdata`**. You can set this option during deployment or installation using the **`--init-config`** installer argument.

You can customize the Greengrass data plane endpoint where the device connects. You can set this configuration option to **`iotdata`** to set the Greengrass data plane endpoint to the same endpoint as the IoT data endpoint, which you can specify with the **`iotDataEndpoint`**.

You must also set this option to **`iotdata`** if you deploy nucleus configuration updates to switch the device's IoT connectivity to a different AWS account or Region. Without this setting, the data plane stays bound to the original account after the connectivity switch. This breaks component artifact downloads and control plane operations.

Configure MQTT timeouts and cache settings

In the AWS IoT Greengrass environment, components can use MQTT to communicate with AWS IoT Core. The AWS IoT Greengrass Core software manages MQTT messages for components. When the core device loses connection to the AWS Cloud, the software caches MQTT messages to retry later when the connection restores. You can configure settings such as message timeouts and the size of the cache. For more information, see the `mqtt` and `mqtt.spooler` configuration parameters of the Greengrass nucleus component.

AWS IoT Core imposes service quotas on its MQTT message broker. These quotas might apply to messages that you send between core devices and AWS IoT Core. For more information, see AWS IoT Core message broker service quotas in the *AWS General Reference*.

Configure Greengrass Nucleus on IPv6 network

Greengrass Nucleus talks to AWS IoT Core through Greengrass APIs. Greengrass APIs support IPv6 under dualstack environment.

To enable dualstack endpoints for IPv6:

- Add system properties `aws.useDualstackEndpoint=true`, and `java.net.preferIPv6Addresses=true` to `jvmOptions`
- Set `s3EndpointType` to `DUALSTACK`

Set this option during deployment, or manually provision it with the `--init-config` installer argument. See Using Amazon S3 dual-stack endpoints for more details.

Example code for deployment:

```
{
    "jvmOptions": "-Daws.useDualstackEndpoint=true",
    "s3EndpointType":"DUALSTACK"
}
```

Example `config.yaml` through manual provisioning:

```
---
system:
  ...
services:
  aws.greengrass.Nucleus:
    ...
    configuration:
      ...
      jvmOptions: "-Daws.useDualstackEndpoint=true -Djava.net.preferIPv6Addresses=true"
      s3EndpointType: "DUALSTACK"
```

Update the AWS IoT Greengrass Core software (OTA)

The AWS IoT Greengrass Core software comprises the Greengrass nucleus component and other optional components that you can deploy to your devices to perform over-the-air (OTA) updates of the software. This feature is built in to the AWS IoT Greengrass Core software.

OTA updates make it more efficient to:

- Fix security vulnerabilities.
- Address software stability issues.

- Deploy new or improved features.

Topics

- Requirements
- Considerations for core devices
- Greengrass nucleus update behavior
- Perform an OTA update

Requirements

The following requirements apply to deploy OTA updates of the AWS IoT Greengrass Core software:

- The Greengrass core device must have a connection to the AWS Cloud to receive the deployment.
- The Greengrass core device must be correctly configured and provisioned with certificates and keys for authentication with AWS IoT Core and AWS IoT Greengrass.
- The AWS IoT Greengrass Core software must be set up and running as a system service. OTA updates don't work if you run the nucleus from the JAR file, `Greengrass.jar`. For more information, see Configure the Greengrass nucleus as a system service.

Considerations for core devices

Before you perform an OTA update, be aware of the impact on the core devices that you update and their connected client devices:

- The Greengrass nucleus shuts down.
- All components running on the core device also shut down. If those components write to local resources, they might leave those resources in an incorrect state unless shut down properly. Components can use interprocess communication to tell the nucleus component to defer the update until they clean up the resources that they use.
- While the nucleus component is shut down, the core device loses its connections with the AWS Cloud and local devices. The core device won't route messages from client devices while shut down.
- Long-lived Lambda functions that run as components lose their dynamic state information and drop all pending work.

Greengrass nucleus update behavior

When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.

When the version of the Greengrass nucleus component changes, the AWS IoT Greengrass Core software—which includes the nucleus and all other components on your device—restarts to apply the changes. Because of the impact on core devices when the nucleus component is updated, you might want to control when a new nucleus patch version is deployed to your devices. To do so, you must directly include the Greengrass nucleus component in your deployment. Directly including a component means that you include a specific version of that component in your deployment configuration and do not rely on component dependencies to deploy that component to your devices. For more information about defining dependencies in your component recipes, see Recipe format.

> **Important**
>
> The AWS IoT Greengrass Core software ignores the failure handling policy for bootstrap deployments. A bootstrap deployment updates the Greengrass nucleus version or includes a component with a bootstrap lifecycle step. If a nucleus update fails, the AWS IoT Greengrass Core software rolls back the core device to its previous configuration. The rollback happens even if you set the failure handling policy to `DO_NOTHING`. This behavior keeps the core device in a working state after a failed update. For more information, see Create deployments.

Review the following table to understand the update behavior for the Greengrass nucleus component based on your actions and deployment configurations.

Action	Deployment configuration	Nucleus update behavior
Add new devices to a thing group targeted by an existing deployment without revising the deployment.	The deployment does not directly include Greengrass nucleus.	On new devices, installs the latest patch version of nucleus that meets all

Action	Deployment configuration	Nucleus update behavior
	The deployment directly includes at least one AWS-provided component, or includes a custom component that depends on an AWS-provided component or on the Greengrass nucleus.	component dependency requirements. On existing devices, does not update the installed version of the nucleus.
Add new devices to a thing group targeted by an existing deployment without revising the deployment.	The deployment directly includes a specific version of the Greengrass nucleus.	On new devices, installs the specified nucleus version. On existing devices, does not update the installed version of the nucleus.
Create a new deployment or revise an existing deployment.	The deployment does not directly include Greengrass nucleus. The deployment directly includes at least one AWS-provided component, or includes a custom component that depends on an AWS-provided component or on the Greengrass nucleus.	On all targeted devices, installs the latest patch version of the nucleus that meets all component dependency requirements, including on any new devices that you add to the targeted thing group.
Create a new deployment or revise an existing deployment.	The deployment directly includes a specific version of the Greengrass nucleus.	On all targeted devices, installs the specified nucleus version, including any new devices that you add to the targeted thing group.

Perform an OTA update

To perform an OTA update, create a deployment that includes the nucleus component and the version to install.

Uninstall the AWS IoT Greengrass Core software

You can uninstall the AWS IoT Greengrass Core software to remove it from a device that you don't want to use as a Greengrass core device. You can also use these steps to clean up an installation that fails.

> **Important**
>
> If you want a device to continue authenticating with AWS IoT but you no longer want AWS IoT Greengrass to bill you for a Greengrass core device, you must complete all the steps in this procedure. Until you complete all the steps, AWS IoT continues to identify the device's certificate as a Greengrass core device. As a result, AWS IoT Greengrass bills you for each month that the device authenticates with AWS IoT. For more information about pricing, see AWS IoT Greengrass pricing.

To uninstall the AWS IoT Greengrass Core software

1. If you run the software as a system service, you must stop, disable, and remove the service. Run the following commands as appropriate for your operating system.

   Linux

   1. Stop the service.

      ```
      sudo systemctl stop greengrass.service
      ```

   2. Disable the service.

      ```
      sudo systemctl disable greengrass.service
      ```

   3. Remove the service.

      ```
      sudo rm /etc/systemd/system/greengrass.service
      ```

4. Verify that the service is deleted.

```
sudo systemctl daemon-reload && sudo systemctl reset-failed
```

Windows (Command Prompt)

> **Note**
>
> You must run Command Prompt as an administrator to run these commands.

1. Stop the service.

```
sc stop "greengrass"
```

2. Disable the service.

```
sc config "greengrass" start=disabled
```

3. Remove the service.

```
sc delete "greengrass"
```

4. Restart the device.

Windows (PowerShell)

> **Note**
>
> You must run PowerShell as an administrator to run these commands.

1. Stop the service.

```
Stop-Service -Name "greengrass"
```

2. Disable the service.

```
Set-Service -Name "greengrass" -Status stopped -StartupType disabled
```

3. Remove the service.

   - For PowerShell 6.0 and later:

     ```
     Remove-Service -Name "greengrass" -Confirm:$false -Verbose
     ```

   - For PowerShell versions earlier than 6.0 :

     ```
     Get-Item HKLM:\SYSTEM\CurrentControlSet\Services\greengrass | Remove-Item
      -Force -Verbose
     ```

4. Restart the device.

2. Remove the root folder from the device. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the root folder.

   Linux

   ```
   sudo rm -rf /greengrass/v2
   ```

   Windows (Command Prompt)

   ```
   rmdir /s /q C:\greengrass\v2
   ```

   Windows (PowerShell)

   ```
   cmd.exe /c "rmdir /s /q C:\greengrass\v2"
   ```

3. Delete the core device from the AWS IoT Greengrass service. This step removes the core device's status information from the AWS Cloud. Be sure to complete this step if you plan to reinstall the AWS IoT Greengrass Core software to a core device with the same name.

   - To delete a core device from the AWS IoT Greengrass console, do the following:

     a. Navigate to the AWS IoT Greengrass console.

     b. Choose **Core devices**.

     c. Choose the core device to delete.

     d. Choose **Delete**.

e. In the confirmation modal, choose **Delete**.

- To delete a core device with the AWS Command Line Interface, use the DeleteCoreDevice operation. Run the following command, and replace *MyGreengrassCore* with the name of the core device.

```
aws greengrassv2 delete-core-device --core-device-thing-name MyGreengrassCore
```

AWS IoT Greengrass V2 tutorials

AWS IoT Greengrass is a service that enables you to run AWS Lambda functions, machine learning models, and other code on edge devices. This allows you to process data locally, reducing latency and bandwidth costs while maintaining secure communication with the cloud.

You can complete the following tutorials to learn about AWS IoT Greengrass V2 and its features.

Topics

- Video tutorials
- Tutorial: Develop a Greengrass component that defers component updates
- Tutorial: Interact with local IoT devices over MQTT
- Tutorial: Secure Greengrass Nucleus with Trusted Platform Module (TPM)
- Tutorial: Secure AWS IoT Greengrass Nucleus Lite with Trusted Platform Module (TPM)
- Tutorial: Get started with SageMaker AI Edge Manager
- Tutorial: Perform sample image classification inference using TensorFlow Lite
- Tutorial: Perform sample image classification inference on images from a camera using TensorFlow Lite
- Tutorial: Set up high availability for AWS IoT Greengrass V2 with Pacemaker
- Switch a Greengrass core device to a different AWS account or Region

Video tutorials

Videos

The following list includes video tutorials for AWS IoT Greengrass V2.

- AWS IoT Greengrass nucleus lite tutorial: Getting started

  April 10, 2025 – In this tutorial, you will learn about Greengrass nucleus lite and when to select it for your AWS IoT devices. You will learn how to install nucleus lite and deploy your components to it.

- AWS IoT Greengrass V2 tutorial: Getting started

December 16, 2021 – Get started with AWS IoT Greengrass V2. In this tutorial, you will learn how to install Greengrass and complete your first deployment.

- AWS IoT Greengrass V2 component development with local debug console

  August 21, 2023 – Learn how to use the Greengrass V2 local debug console to develop and debug your components quickly.

- Connect MQTT client devices to AWS IoT Greengrass V2

  August 2, 2023 – In this tutorial, you will learn about how to configure a core device to interact with local AWS IoT devices that connect to the core device over MQTT.

- AWS IoT Greengrass V2 tutorial: Using MQTT applications

  December 16, 2021 – Create, test, and deploy an AWS IoT Greengrass V2 native component to publish and subscribe to MQTT through AWS IoT Core.

Tutorial: Develop a Greengrass component that defers component updates

You can complete this tutorial to develop a component that defers over-the-air deployment updates. When you deploy updates to your devices, you might want to delay updates based on conditions, such as the following:

- The device has a low battery level.
- The device is running a process or job that can't be interrupted.
- The device has a limited or expensive internet connection.

> **Note**
>
> A *component* is a software module that runs on AWS IoT Greengrass core devices. Components enable you to create and manage complex applications as discrete building blocks that you can reuse from one Greengrass core device to another.

In this tutorial, you do the following:

1. Install the Greengrass Development Kit CLI (GDK CLI) on your development computer. The GDK CLI provides features that help you develop custom Greengrass components.
2. Develop a Hello World component that defers component updates when the core device's battery level is below a threshold. This component subscribes to update notifications using the SubscribeToComponentUpdates IPC operation. When it receives the notification, it checks if the battery level is lower than a customizable threshold. If the battery level is below the threshold, it defers the update for 30 seconds using the DeferComponentUpdate IPC operation. You develop this component on your development computer using the GDK CLI.

   > **Note**
   >
   > This component reads battery level from a file that you create on the core device to imitate a real battery, so you can complete this tutorial on a core device without a battery.

3. Publish that component to the AWS IoT Greengrass service.
4. Deploy that component from the AWS Cloud to a Greengrass core device to test it. Then, you modify the virtual battery level on the core device, and create additional deployments to see how the core device defers updates when the battery level is low.

You can expect to spend 20–30 minutes on this tutorial.

Prerequisites

To complete this tutorial, you need the following:

- An AWS account. If you don't have one, see Set up an AWS account.
- An AWS Identity and Access Management (IAM) user with administrator permissions.
- A Greengrass core device with an internet connection. For more information about how to set up a core device, see Setting up AWS IoT Greengrass core devices.
  - Python 3.6 or later installed for all users on the core device and added to the PATH environment variable. On Windows, you must also have the Python Launcher for Windows installed for all users.

> **Important**
>
> In Windows, Python doesn't install for all users by default. When you install Python, you must customize the installation to configure it for the AWS IoT Greengrass Core software to run Python scripts. For example, if you use the graphical Python installer, do the following:
>
> 1. Select **Install launcher for all users (recommended)**.
> 2. Choose **Customize installation**.
> 3. Choose **Next**.
> 4. Select **Install for all users**.
> 5. Select **Add Python to environment variables**.
> 6. Choose **Install**.
>
> For more information, see Using Python on Windows in the *Python 3 documentation*.

- A Windows, macOS, or Unix-like development computer with an internet connection.
  - Python 3.6 or later installed on your development computer.
  - Git installed on your development computer.
  - AWS Command Line Interface (AWS CLI) installed and configured with credentials on your development computer. For more information, see Installing, updating, and uninstalling the AWS CLI and Configuring the AWS CLI in the *AWS Command Line Interface User Guide*.

> **Note**
>
> If you use a Raspberry Pi or another 32-bit ARM device, install AWS CLI V1. AWS CLI V2 isn't available for 32-bit ARM devices. For more information, see Installing, updating, and uninstalling the AWS CLI version 1.

Step 1: Install the Greengrass Development Kit CLI

The Greengrass Development Kit CLI (GDK CLI) provides features that help you develop custom Greengrass components. You can use the GDK CLI to create, build, and publish custom components.

If you haven't installed the GDK CLI on your development computer, complete the following steps to install it.

To install the latest version of the GDK CLI

1. On your development computer, run the following command to install the latest version of the GDK CLI from its GitHub repository.

   ```
   python3 -m pip install -U git+https://github.com/aws-greengrass/aws-greengrass-gdk-cli.git@v1.6.2
   ```

2. Run the following command to verify that the GDK CLI installed successfully.

   ```
   gdk --help
   ```

   If the gdk command isn't found, add its folder to PATH.

   - On Linux devices, add /home/*MyUser*/.local/bin to PATH, and replace *MyUser* with the name of your user.
   - On Windows devices, add *PythonPath*\\Scripts to PATH, and replace *PythonPath* with the path to the Python folder on your device.

Step 2: Develop a component that defers updates

In this section, you develop a Hello World component in Python that defers component updates when the core device's battery level is below a threshold that you configure when you deploy the component. In this component, you use the interprocess communication (IPC) interface in the AWS IoT Device SDK v2 for Python. You use the SubscribeToComponentUpdates IPC operation to receive notifications when the core device receives a deployment. Then, you use the DeferComponentUpdate IPC operation to defer or acknowledge the update based on the device's battery level.

To develop a Hello World component that defers updates

1. On your development computer, create a folder for the component source code.

   ```
   mkdir com.example.BatteryAwareHelloWorld
   cd com.example.BatteryAwareHelloWorld
   ```

2. Use a text editor to create a file named `gdk-config.json`. The GDK CLI reads from the GDK CLI configuration file, named `gdk-config.json`, to build and publish components. This configuration file exists in the root of the component folder.

   For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

   ```
   nano gdk-config.json
   ```

   Copy the following JSON into the file.

   - Replace *Amazon* with your name.
   - Replace *us-west-2* with the AWS Region where your core device operates. The GDK CLI publishes the component in this AWS Region.
   - Replace *greengrass-component-artifacts* with the S3 bucket prefix to use. When you use the GDK CLI to publish the component, the GDK CLI uploads the component's artifacts to the S3 bucket whose name is formed from this value, the AWS Region, and your AWS account ID using the following format: *bucketPrefix*-*region*-*accountId*.

     For example, if you specify **greengrass-component-artifacts** and **us-west-2**, and your AWS account ID is **123456789012**, the GDK CLI uses the S3 bucket named `greengrass-component-artifacts-us-west-2-123456789012`.

   ```
   {
     "component": {
       "com.example.BatteryAwareHelloWorld": {
         "author": "Amazon",
         "version": "NEXT_PATCH",
         "build": {
           "build_system" : "zip"
         },
         "publish": {
           "region": "us-west-2",
           "bucket": "greengrass-component-artifacts"
         }
       }
     },
     "gdk_version": "1.0.0"
   ```

```
}
```

The configuration file specifies the following:

- The version to use when the GDK CLI publishes the Greengrass component to the AWS IoT Greengrass cloud service. `NEXT_PATCH` specifies to choose the next patch version after the latest version available in the AWS IoT Greengrass cloud service. If the component doesn't have a version in the AWS IoT Greengrass cloud service yet, the GDK CLI uses `1.0.0`.
- The build system for the component. When you use the `zip` build system, the GDK CLI packages the component's source into a ZIP file that becomes the component's single artifact.
- The AWS Region where the GDK CLI publishes the Greengrass component.
- The prefix for the S3 bucket where the GDK CLI uploads the component's artifacts.

3. Use a text editor to create the component source code in a file named `main.py`.

   For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

   ```
   nano main.py
   ```

   Copy the following Python code into the file.

   ```
   import json
   import os
   import sys
   import time
   import traceback

   from pathlib import Path

   from awsiot.greengrasscoreipc.clientv2 import GreengrassCoreIPCClientV2

   HELLO_WORLD_PRINT_INTERVAL = 15  # Seconds
   DEFER_COMPONENT_UPDATE_INTERVAL = 30 * 1000  # Milliseconds


   class BatteryAwareHelloWorldPrinter():
       def __init__(self, ipc_client: GreengrassCoreIPCClientV2, battery_file_path:
    Path, battery_threshold: float):
   ```

```
        self.battery_file_path = battery_file_path
        self.battery_threshold = battery_threshold
        self.ipc_client = ipc_client
        self.subscription_operation = None

    def on_component_update_event(self, event):
        try:
            if event.pre_update_event is not None:
                if self.is_battery_below_threshold():
                    self.defer_update(event.pre_update_event.deployment_id)
                    print('Deferred update for deployment %s' %
                          event.pre_update_event.deployment_id)
                else:
                    self.acknowledge_update(
                        event.pre_update_event.deployment_id)
                    print('Acknowledged update for deployment %s' %
                          event.pre_update_event.deployment_id)
            elif event.post_update_event is not None:
                print('Applied update for deployment')
        except:
            traceback.print_exc()

    def subscribe_to_component_updates(self):
        if self.subscription_operation == None:
            # SubscribeToComponentUpdates returns a tuple with the response and the
operation.
            _, self.subscription_operation =
self.ipc_client.subscribe_to_component_updates(
                on_stream_event=self.on_component_update_event)

    def close_subscription(self):
        if self.subscription_operation is not None:
            self.subscription_operation.close()
            self.subscription_operation = None

    def defer_update(self, deployment_id):
        self.ipc_client.defer_component_update(
            deployment_id=deployment_id,
recheck_after_ms=DEFER_COMPONENT_UPDATE_INTERVAL)

    def acknowledge_update(self, deployment_id):
        # Specify recheck_after_ms=0 to acknowledge a component update.
        self.ipc_client.defer_component_update(
            deployment_id=deployment_id, recheck_after_ms=0)
```

```
    def is_battery_below_threshold(self):
        return self.get_battery_level() < self.battery_threshold

    def get_battery_level(self):
        # Read the battery level from the virtual battery level file.
        with self.battery_file_path.open('r') as f:
            data = json.load(f)
            return float(data['battery_level'])

    def print_message(self):
        message = 'Hello, World!'
        if self.is_battery_below_threshold():
            message += ' Battery level (%d) is below threshold (%d), so the
 component will defer updates' % (
                self.get_battery_level(), self.battery_threshold)
        else:
            message += ' Battery level (%d) is above threshold (%d), so the
 component will acknowledge updates' % (
                self.get_battery_level(), self.battery_threshold)
        print(message)


def main():
    # Read the battery threshold and virtual battery file path from command-line
 args.
    args = sys.argv[1:]
    battery_threshold = float(args[0])
    battery_file_path = Path(args[1])
    print('Reading battery level from %s and deferring updates when below %d' % (
        str(battery_file_path), battery_threshold))

    try:
        # Create an IPC client and a Hello World printer that defers component
 updates.
        ipc_client = GreengrassCoreIPCClientV2()
        hello_world_printer = BatteryAwareHelloWorldPrinter(
            ipc_client, battery_file_path, battery_threshold)
        hello_world_printer.subscribe_to_component_updates()
        try:
            # Keep the main thread alive, or the process will exit.
            while True:
                hello_world_printer.print_message()
                time.sleep(HELLO_WORLD_PRINT_INTERVAL)
```

```
        except InterruptedError:
            print('Subscription interrupted')
        hello_world_printer.close_subscription()
    except Exception:
        print('Exception occurred', file=sys.stderr)
        traceback.print_exc()
        exit(1)


if __name__ == '__main__':
    main()
```

This Python application does the following:

- Reads the core device's battery level from a virtual battery level file that you'll create on the core device later. This virtual battery level file imitates a real battery, so you can complete this tutorial on core devices that don't have a battery.
- Reads command-line arguments for the battery threshold and the path to the virtual battery level file. The component recipe sets these command-line arguments based on configuration parameters, so you can customize these values when you deploy the component.
- Uses the IPC client V2 in the AWS IoT Device SDK v2 for Python to communicate with the AWS IoT Greengrass Core software. Compared to the original IPC client, the IPC client V2 reduces the amount of code that you need to write to use IPC in custom components.
- Subscribes to update notifications using the SubscribeToComponentUpdates IPC operation. The AWS IoT Greengrass Core software sends notifications before and after each deployment. The component calls the following function each time it receives a notification. If the notification is for an upcoming deployment, the component checks if the battery level is lower than a threshold. If the battery level is below the threshold, the component defers the update for 30 seconds using the DeferComponentUpdate IPC operation. Otherwise, if the battery level isn't below the threshold, the component acknowledges the update, so the update can proceed.

  ```
  def on_component_update_event(self, event):
      try:
          if event.pre_update_event is not None:
              if self.is_battery_below_threshold():
                  self.defer_update(event.pre_update_event.deployment_id)
                  print('Deferred update for deployment %s' %
  ```

```
                    event.pre_update_event.deployment_id)
            else:
                self.acknowledge_update(
                    event.pre_update_event.deployment_id)
                print('Acknowledged update for deployment %s' %
                      event.pre_update_event.deployment_id)
        elif event.post_update_event is not None:
            print('Applied update for deployment')
    except:
        traceback.print_exc()
```

> **Note**
>
> The AWS IoT Greengrass Core software doesn't send update notifications for local deployments, so you deploy this component using the AWS IoT Greengrass cloud service to test it.

4. Use a text editor to create the component recipe in a file named `recipe.json` or `recipe.yaml`. The component *recipe* defines the component's metadata, default configuration parameters, and platform-specific lifecycle scripts.

   JSON

   For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

   ```
   nano recipe.json
   ```

   Copy the following JSON into the file.

   ```
   {
     "RecipeFormatVersion": "2020-01-25",
     "ComponentName": "COMPONENT_NAME",
     "ComponentVersion": "COMPONENT_VERSION",
     "ComponentDescription": "This Hello World component defers updates when the
    battery level is below a threshold.",
     "ComponentPublisher": "COMPONENT_AUTHOR",
     "ComponentConfiguration": {
       "DefaultConfiguration": {
         "BatteryThreshold": 50,
         "LinuxBatteryFilePath": "/home/ggc_user/virtual_battery.json",
   ```

```
      "WindowsBatteryFilePath": "C:\\Users\\ggc_user\\virtual_battery.json"
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "install": "python3 -m pip install --user awsiotsdk --upgrade",
        "Run": "python3 -u {artifacts:decompressedPath}/
com.example.BatteryAwareHelloWorld/main.py \"{configuration:/BatteryThreshold}\"
 \"{configuration:/LinuxBatteryFilePath}\""
      },
      "Artifacts": [
        {
          "Uri": "s3://BUCKET_NAME/COMPONENT_NAME/COMPONENT_VERSION/
com.example.BatteryAwareHelloWorld.zip",
          "Unarchive": "ZIP"
        }
      ]
    },
    {
      "Platform": {
        "os": "windows"
      },
      "Lifecycle": {
        "install": "py -3 -m pip install --user awsiotsdk --upgrade",
        "Run": "py -3 -u {artifacts:decompressedPath}/
com.example.BatteryAwareHelloWorld/main.py \"{configuration:/BatteryThreshold}\"
 \"{configuration:/WindowsBatteryFilePath}\""
      },
      "Artifacts": [
        {
          "Uri": "s3://BUCKET_NAME/COMPONENT_NAME/COMPONENT_VERSION/
com.example.BatteryAwareHelloWorld.zip",
          "Unarchive": "ZIP"
        }
      ]
    }
  ]
}
```

YAML

For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

```
nano recipe.yaml
```

Copy the following YAML into the file.

```
---
RecipeFormatVersion: "2020-01-25"
ComponentName: "COMPONENT_NAME"
ComponentVersion: "COMPONENT_VERSION"
ComponentDescription: "This Hello World component defers updates when the
 battery level is below a threshold."
ComponentPublisher: "COMPONENT_AUTHOR"
ComponentConfiguration:
  DefaultConfiguration:
    BatteryThreshold: 50
    LinuxBatteryFilePath: "/home/ggc_user/virtual_battery.json"
    WindowsBatteryFilePath: "C:\\Users\\ggc_user\\virtual_battery.json"
Manifests:
  - Platform:
      os: linux
    Lifecycle:
      install: python3 -m pip install --user awsiotsdk --upgrade
      Run: python3 -u {artifacts:decompressedPath}/
com.example.BatteryAwareHelloWorld/main.py "{configuration:/BatteryThreshold}"
 "{configuration:/LinuxBatteryFilePath}"
    Artifacts:
      - Uri: "s3://BUCKET_NAME/COMPONENT_NAME/COMPONENT_VERSION/
com.example.BatteryAwareHelloWorld.zip"
        Unarchive: ZIP
  - Platform:
      os: windows
    Lifecycle:
      install: py -3 -m pip install --user awsiotsdk --upgrade
      Run: py -3 -u {artifacts:decompressedPath}/
com.example.BatteryAwareHelloWorld/main.py "{configuration:/BatteryThreshold}"
 "{configuration:/WindowsBatteryFilePath}"
    Artifacts:
```

```
      - Uri: "s3://BUCKET_NAME/COMPONENT_NAME/COMPONENT_VERSION/
com.example.BatteryAwareHelloWorld.zip"
        Unarchive: ZIP
```

This recipe specifies the following:

- Default configuration parameters for the battery threshold, the virtual battery file path on Linux core devices, and the virtual battery file path on Windows core devices.
- An `install` lifecycle that installs the latest version of the AWS IoT Device SDK v2 for Python.
- A `run` lifecycle that runs the Python application in `main.py`.
- Placeholders, such as `COMPONENT_NAME` and `COMPONENT_VERSION`, where the GDK CLI replaces information when it builds the component recipe.

For more information about component recipes, see AWS IoT Greengrass component recipe reference.

Step 3: Publish the component to the AWS IoT Greengrass service

In this section, you publish the Hello World component to the AWS IoT Greengrass cloud service. After a component is available in the AWS IoT Greengrass cloud service, you can deploy it to core devices. You use the GDK CLI to publish the component from your development computer to the AWS IoT Greengrass cloud service. The GDK CLI uploads the component's recipe and artifacts for you.

To publish the Hello World component to the AWS IoT Greengrass service

1. Run the following command to build the component using the GDK CLI. The component build command creates a recipe and artifacts based on the GDK CLI configuration file. In this process, the GDK CLI creates a ZIP file that contains the component's source code.

   ```
   gdk component build
   ```

   You should see messages similar to the following example.

   ```
   [2022-04-28 11:20:16] INFO - Getting project configuration from gdk-config.json
   ```

```
[2022-04-28 11:20:16] INFO - Found component recipe file 'recipe.yaml' in the
 project directory.
[2022-04-28 11:20:16] INFO - Building the component
 'com.example.BatteryAwareHelloWorld' with the given project configuration.
[2022-04-28 11:20:16] INFO - Using 'zip' build system to build the component.
[2022-04-28 11:20:16] WARNING - This component is identified as using 'zip' build
 system. If this is incorrect, please exit and specify custom build command in the
 'gdk-config.json'.
[2022-04-28 11:20:16] INFO - Zipping source code files of the component.
[2022-04-28 11:20:16] INFO - Copying over the build artifacts to the greengrass
 component artifacts build folder.
[2022-04-28 11:20:16] INFO - Updating artifact URIs in the recipe.
[2022-04-28 11:20:16] INFO - Creating component recipe in 'C:\Users\finthomp
\greengrassv2\com.example.BatteryAwareHelloWorld\greengrass-build\recipes'.
```

2. Run the following command to publish the component to the AWS IoT Greengrass cloud service. The component publish command uploads the component's ZIP file artifact to an S3 bucket. Then, it updates the ZIP file's S3 URI in the component recipe and uploads the recipe to the AWS IoT Greengrass service. In this process, the GDK CLI checks what version of the Hello World component is already available in the AWS IoT Greengrass cloud service, so it can choose the next patch version after that version. If the component doesn't exist yet, the GDK CLI uses version `1.0.0`.

   ```
   gdk component publish
   ```

   You should see messages similar to the following example. The output tells you the version of the component that the GDK CLI created.

   ```
   [2022-04-28 11:20:29] INFO - Getting project configuration from gdk-config.json
   [2022-04-28 11:20:29] INFO - Found component recipe file 'recipe.yaml' in the
    project directory.
   [2022-04-28 11:20:29] INFO - Found credentials in shared credentials file: ~/.aws/
   credentials
   [2022-04-28 11:20:30] INFO - No private version of the component
    'com.example.BatteryAwareHelloWorld' exist in the account. Using '1.0.0' as the
    next version to create.
   [2022-04-28 11:20:30] INFO - Publishing the component
    'com.example.BatteryAwareHelloWorld' with the given project configuration.
   [2022-04-28 11:20:30] INFO - Uploading the component built artifacts to s3 bucket.
   [2022-04-28 11:20:30] INFO - Uploading component artifacts to S3
    bucket: greengrass-component-artifacts-us-west-2-123456789012. If this is your
   ```

```
 first time using this bucket, add the 's3:GetObject' permission to each core
 device's token exchange role to allow it to download the component artifacts. For
 more information, see https://docs.aws.amazon.com/greengrass/v2/developerguide/
device-service-role.html.
[2022-04-28 11:20:30] INFO - Not creating an artifacts bucket as it already exists.
[2022-04-28 11:20:30] INFO - Updating the component recipe
 com.example.BatteryAwareHelloWorld-1.0.0.
[2022-04-28 11:20:31] INFO - Creating a new greengrass component
 com.example.BatteryAwareHelloWorld-1.0.0
[2022-04-28 11:20:31] INFO - Created private version '1.0.0' of the component in
 the account.'com.example.BatteryAwareHelloWorld'.
```

3. Copy the S3 bucket name from the output. You use the bucket name later to allow the core device to download component artifacts from this bucket.
4. (Optional) View the component in the AWS IoT Greengrass console to verify that it uploaded successfully. Do the following:

   a. In the AWS IoT Greengrass console navigation menu, choose **Components**.

   b. On the **Components** page, choose the **My components** tab, and then choose **com.example.BatteryAwareHelloWorld**.

      On this page, you can see the component's recipe and other information about the component.
5. Allow the core device to access component artifacts in the S3 bucket.

   Each core device has a core device IAM role that allows it to interact with AWS IoT and send logs to the AWS Cloud. This device role doesn't allow access to S3 buckets by default, so you must create and attach a policy that allows the core device to retrieve component artifacts from the S3 bucket.

   If your device's role already allows access to the S3 bucket, you can skip this step. Otherwise, create an IAM policy that allows access and attach it to the role, as follows:

   a. In the IAM console navigation menu, choose **Policies**, and then choose **Create policy**.

   b. On the **JSON** tab, replace the placeholder content with the following policy. Replace *greengrass-component-artifacts-us-west-2-123456789012* with the name of the S3 bucket where the GDK CLI uploaded the component's artifacts.

      For example, if you specified **greengrass-component-artifacts** and **us-west-2** in the GDK CLI configuration file, and your AWS account ID is **123456789012**, the

GDK CLI uses the S3 bucket named `greengrass-component-artifacts-us-west-2-123456789012`.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetObject"
      ],
      "Resource": "arn:aws:s3:::greengrass-component-artifacts-us-west-2-123456789012/*"
    }
  ]
}
```

c. Choose **Next**.

d. In the **Policy details section**, for **Name**, enter **`MyGreengrassV2ComponentArtifactPolicy`**.

e. Choose **Create policy**.

f. In the IAM console navigation menu, choose **Role**, and then choose the name of the role for the core device. You specified this role name when you installed the AWS IoT Greengrass Core software. If you did not specify a name, the default is `GreengrassV2TokenExchangeRole`.

g. Under **Permissions**, choose **Add permissions**, then choose **Attach policies**.

h. On the **Add permissions** page, select the check box next to the `MyGreengrassV2ComponentArtifactPolicy` policy that you created, and then choose **Add permissions**.

Step 4: Deploy and test the component on a core device

In this section, you deploy the component to the core device to test its functionality. On the core device, you create the virtual battery level file to imitate a real battery. Then, you create additional

deployments and observe the component log files on the core device to see the component defer and acknowledge updates.

To deploy and test the Hello World component that defers updates

1. Use a text editor to create a virtual battery level file. This file imitates a real battery.

   - On Linux core devices, create a file named `/home/ggc_user/virtual_battery.json`. Run the text editor with `sudo` permissions.
   - On Windows core devices, create a file named `C:\Users\ggc_user\virtual_battery.json`. Run the text editor as an administrator.

   For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

   ```
   sudo nano /home/ggc_user/virtual_battery.json
   ```

   Copy the following JSON into the file.

   ```
   {
     "battery_level": 50
   }
   ```

2. Deploy the Hello World component to the core device. Do the following:

   a. In the AWS IoT Greengrass console navigation menu, choose **Components**.

   b. On the **Components** page, choose the **My components** tab, and then choose **com.example.BatteryAwareHelloWorld**.

   c. On the **com.example.BatteryAwareHelloWorld** page, choose **Deploy**.

   d. From **Add to deployment**, choose an existing deployment to revise, or choose to create a new deployment, and then choose **Next**.

   e. If you chose to create a new deployment, choose the target core device or thing group for the deployment. On the **Specify target** page, under **Deployment target**, choose a core device or thing group, and then choose **Next**.

   f. On the **Select components** page, verify that the **com.example.BatteryAwareHelloWorld** component is selected, choose **Next**.

g. On the **Configure components** page, select **com.example.BatteryAwareHelloWorld**, and then do the following:

i. Choose **Configure component**.

ii. In the **Configure com.example.BatteryAwareHelloWorld** modal, under **Configuration update**, in **Configuration to merge**, enter the following configuration update.

```
{
  "BatteryThreshold": 70
}
```

iii. Choose **Confirm** to close the modal, and then choose **Next**.

h. On the **Confirm advanced settings** page, in the **Deployment policies** section, under **Component update policy**, confirm that **Notify components** is selected. **Notify components** is selected by default when you create a new deployment.

i. On the **Review** page, choose **Deploy**.

The deployment can take up to a minute to complete.

3. The AWS IoT Greengrass Core software saves stdout from component processes to log files in the `logs` folder. Run the following command to verify that the Hello World component runs and prints status messages.

Linux or Unix

```
sudo tail -f /greengrass/v2/logs/com.example.BatteryAwareHelloWorld.log
```

Windows Command Prompt (CMD)

```
type C:\greengrass\v2\logs\com.example.BatteryAwareHelloWorld.log
```

PowerShell

```
gc C:\greengrass\v2\logs\com.example.BatteryAwareHelloWorld.log -Tail 10 -Wait
```

You should see messages similar to the following example.

```
Hello, World! Battery level (50) is below threshold (70), so the component will
 defer updates.
```

Note

If the file doesn't exist, the deployment may not be complete yet. If the file doesn't exist within 30 seconds, the deployment likely failed. This can occur if the core device doesn't have permission to download the component's artifacts from the S3 bucket, for example. Run the following command to view the AWS IoT Greengrass Core software log file. This file includes logs from the Greengrass core device's deployment service.

Linux or Unix

```
sudo tail -f /greengrass/v2/logs/greengrass.log
```

Windows Command Prompt (CMD)

```
type C:\greengrass\v2\logs\greengrass.log
```

The `type` command writes the file's contents to the terminal. Run this command multiple times to observe changes in the file.

PowerShell

```
gc C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
```

4. Create a new deployment to the core device to verify that the component defers the update. Do the following:

   a. In the AWS IoT Greengrass console navigation menu, choose **Deployments**.

   b. Choose the deployment that you created or revised earlier.

   c. On the deployment page, choose **Revise**.

   d. In the **Revise deployment** modal, choose **Revise deployment**.

   e. Choose **Next** at each step, and then choose **Deploy**.

5. Run the following command to view the component's logs again, and verify that it defers the update.

Linux or Unix

```
sudo tail -f /greengrass/v2/logs/com.example.BatteryAwareHelloWorld.log
```

Windows Command Prompt (CMD)

```
type C:\greengrass\v2\logs\com.example.BatteryAwareHelloWorld.log
```

PowerShell

```
gc C:\greengrass\v2\logs\com.example.BatteryAwareHelloWorld.log -Tail 10 -Wait
```

You should see messages similar to the following example. The component defers the update for 30 seconds, so the component prints this message repeatedly.

```
Deferred update for deployment 50722a95-a05f-4e2a-9414-da80103269aa.
```

6. Use a text editor to edit the virtual battery level file and change the battery level to a value above the threshold, so the deployment can proceed.

   - On Linux core devices, edit the file named `/home/ggc_user/virtual_battery.json`. Run the text editor with `sudo` permissions.
   - On Windows core devices, edit the file named `C:\Users\ggc_user\virtual_battery.json`. Run the text editor as an administrator.

   For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

   ```
   sudo nano /home/ggc_user/virtual_battery.json
   ```

   Change the battery level to `80`.

   ```
   {
     "battery_level": 80
   }
   ```

7. Run the following command to view the component's logs again, and verify that it acknowledges the update.

   Linux or Unix

   ```
   sudo tail -f /greengrass/v2/logs/com.example.BatteryAwareHelloWorld.log
   ```

   Windows Command Prompt (CMD)

   ```
   type C:\greengrass\v2\logs\com.example.BatteryAwareHelloWorld.log
   ```

   PowerShell

   ```
   gc C:\greengrass\v2\logs\com.example.BatteryAwareHelloWorld.log -Tail 10 -Wait
   ```

   You should see messages similar to the following examples.

   ```
   Hello, World! Battery level (80) is above threshold (70), so the component will
    acknowledge updates.
   Acknowledged update for deployment f9499eb2-4a40-40a7-86c1-c89887d859f1.
   ```

You've completed this tutorial. The Hello World component defers or acknowledges updates based on the core device's battery level. For more information about the topics that this tutorial explores, see the following:

- Develop AWS IoT Greengrass components
- Deploy AWS IoT Greengrass components to devices
- Use the AWS IoT Device SDK to communicate with the Greengrass nucleus, other components, and AWS IoT Core
- AWS IoT Greengrass Development Kit Command-Line Interface

Tutorial: Interact with local IoT devices over MQTT

You can complete this tutorial to configure a core device to interact with local IoT devices, called *client devices*, that connect to the core device over MQTT. In this tutorial, you configure AWS IoT things to use *cloud discovery* to connect to the core device as client devices. When you configure

cloud discovery, a client device can send a request to the AWS IoT Greengrass cloud service to discover core devices. The response from AWS IoT Greengrass includes connectivity information and certificates for the core devices that you configure the client device to discover. Then, the client device can use this information to connect to an available core device where it can communicate over MQTT.

In this tutorial, you do the following:

1. Review and update the core device's permissions, if needed.
2. Associate client devices to the core device, so they can discover the core device using cloud discovery.
3. Deploy Greengrass components to the core device to enable client device support.
4. Connect client devices to the core device and test communication with the AWS IoT Core cloud service.
5. Develop a custom Greengrass component that communicates with the client devices.
6. Develop a custom component that interacts with the client devices' AWS IoT device shadows.

This tutorial uses a single core device and a single client device. You can also follow the tutorial to connect and test multiple client devices.

You can expect to spend 30–60 minutes on this tutorial.

Prerequisites

To complete this tutorial, you need the following:

- An AWS account. If you don't have one, see Set up an AWS account.
- An AWS Identity and Access Management (IAM) user with administrator permissions.
- A Greengrass core device. For more information about how to set up a core device, see Setting up AWS IoT Greengrass core devices.
  - The core device must run Greengrass nucleus v2.6.0 or later. This version includes support for wildcards in local publish/subscribe communication and support for client device shadows.

    Note

    Client device support requires Greengrass nucleus v2.2.0 or later. However, this tutorial explores newer features, such as support for MQTT wildcards in local publish/subscribe

and support for client device shadows. These features require Greengrass nucleus v2.6.0 or later.

  - The core device must be on the same network as the client devices to connect.
  - (Optional) To complete the modules where you develop custom Greengrass components, the core device must run the Greengrass CLI. For more information, see Install the Greengrass CLI.
- An AWS IoT thing to connect as a client device in this tutorial. For more information, see Create AWS IoT resources in the *AWS IoT Core Developer Guide*.
  - The client device's AWS IoT policy must allow the `greengrass:Discover` permission. For more information, see Minimal AWS IoT policy for client devices.
  - The client device must be on the same network as the core device.
  - The client device must run Python 3.
  - The client device must run Git.

Step 1: Review and update the core device AWS IoT policy

To support client devices, a core device's AWS IoT policy must allow the following permissions:

- `greengrass:PutCertificateAuthorities`
- `greengrass:VerifyClientDeviceIdentity`
- `greengrass:VerifyClientDeviceIoTCertificateAssociation`
- `greengrass:GetConnectivityInfo`
- `greengrass:UpdateConnectivityInfo` – (Optional) This permission is required to use the IP detector component, which reports the core device's network connectivity information to the AWS IoT Greengrass cloud service.

For more information about these permissions and AWS IoT policies for core devices, see AWS IoT policies for data plane operations and Minimal AWS IoT policy to support client devices.

In this section, you review the AWS IoT policies for your core device and add any required permissions that are missing. If you used the AWS IoT Greengrass Core software installer to provision resources, your core device has an AWS IoT policy that allows access to all AWS IoT Greengrass actions (`greengrass:*`). In this case, you must update the AWS IoT policy only if you plan to configure the shadow manager component to sync device shadows with AWS IoT Core. Otherwise, you can skip this section.

To review and update a core device's AWS IoT policy

1. In the AWS IoT Greengrass console navigation menu, choose **Core devices**.
2. On the **Core devices** page, choose the core device to update.
3. On the core device details page, choose the link to the core device's **Thing**. This link opens the thing details page in the AWS IoT console.
4. On the thing details page, choose **Certificates**.
5. In the **Certificates** tab, choose the thing's active certificate.
6. On the certificate details page, choose **Policies**.
7. In the **Policies** tab, choose the AWS IoT policy to review and update. You can add the required permissions to any policy that is attached to the core device's active certificate.

   > **Note**
   >
   > If you used the AWS IoT Greengrass Core software installer to provision resources, you have two AWS IoT policies. We recommend that you choose the policy named **GreengrassV2IoTThingPolicy**, if it exists. Core devices that you create with the quick installer use this policy name by default. If you add permissions to this policy, you are also granting these permissions to other core devices that use this policy.

8. In the policy overview, choose **Edit active version**.
9. Review the policy for the required permissions, and add any required permissions that are missing.
10. To set a new policy version as the active version, under **Policy version status**, select **Set the edited version as the active version for this policy**.
11. Choose **Save as new version**.

Step 2: Enable client device support

For a client device to use cloud discovery to connect to a core device, you must associate the devices. When you associate a client device to a core device, you enable that client device to retrieve the core device's IP addresses and certificates to use to connect.

To enable client devices to securely connect to a core device and communicate with Greengrass components and AWS IoT Core, you deploy the following Greengrass components to the core device:

- Client device auth (`aws.greengrass.clientdevices.Auth`)

  Deploy the client device auth component to authenticate client devices and authorize client device actions. This component allows your AWS IoT things to connect to a core device.

  This component requires some configuration to use it. You must specify groups of client devices and the operations that each group is authorized to perform, such as to connect and communicate over MQTT. For more information, see client device auth component configuration.

- MQTT 3.1.1 broker (Moquette) (`aws.greengrass.clientdevices.mqtt.Moquette`)

  Deploy the Moquette MQTT broker component to run a lightweight MQTT broker. The Moquette MQTT broker is compliant with MQTT 3.1.1 and includes local support for QoS 0, QoS 1, QoS 2, retained messages, last will messages, and persistent subscriptions.

  You aren't required to configure this component to use it. However, you can configure the port where this component operates the MQTT broker. By default, it uses port 8883.

- MQTT bridge (`aws.greengrass.clientdevices.mqtt.Bridge`)

  (Optional) Deploy the MQTT bridge component to relay messages between client devices (local MQTT), local publish/subscribe, and AWS IoT Core MQTT. Configure this component to sync client devices with AWS IoT Core and interact with client devices from Greengrass components.

  This component requires configuration to use. You must specify the topic mappings where this component relays messages. For more information, see MQTT bridge component configuration.

- IP detector (`aws.greengrass.clientdevices.IPDetector`)

  (Optional) Deploy the IP detector component to automatically report the core device's MQTT broker endpoints to the AWS IoT Greengrass cloud service. You cannot use this component if you have a complex network setup, such as one where a router forwards the MQTT broker port to the core device.

  You aren't required to configure this component to use it.

In this section, you use the AWS IoT Greengrass console to associate client devices and deploy client device components to a core device.

To enable client device support

1. Navigate to the AWS IoT Greengrass console.

2. In the left navigation menu, choose **Core devices**.
3. On the **Core devices** page, choose the core device where you want to enable client device support.
4. On the core device details page, choose the **Client devices** tab.
5. On the **Client devices** tab, choose **Configure cloud discovery**.

   The **Configure core device discovery** page opens. On this page, you can associate client devices to a core device and deploy client device components. This page selects the core device for you in **Step 1: Select target core devices**.

   > **Note**
   >
   > You can also use this page to configure core device discovery for a thing group. If you choose this option, you can deploy client device components to all core devices in a thing group. However, if you choose this option, you must manually associate client devices to each core device later after you create the deployment. In this tutorial, you configure a single core device.

6. In **Step 2: Associate client devices**, associate the client device's AWS IoT thing to the core device. This enables the client device to use cloud discovery to retrieve the core device's connectivity information and certificates. Do the following:

   a. Choose **Associate client devices**.
   b. In the **Associate client devices with core device** modal, enter the name of the AWS IoT thing to associate.
   c. Choose **Add**.
   d. Choose **Associate**.

7. In **Step 3: Configure and deploy Greengrass components**, deploy components to enable client device support. If the target core device has a previous deployment, this page revises that deployment. Otherwise, this page creates a new deployment for the core device. Do the following to configure and deploy the client device components:

   a. The core device must run Greengrass nucleus v2.6.0 or later to complete this tutorial. If the core device runs an earlier version, do the following:

      i. Select the box to deploy the **aws.greengrass.Nucleus** component.
      ii. For the **aws.greengrass.Nucleus** component, choose **Edit configuration**.

iii. For **Component version**, choose version 2.6.0 or later.

iv. Choose **Confirm**.

> **Note**
>
> If you upgrade the Greengrass nucleus from an earlier minor version, and the core device runs AWS-provided components that depend on the nucleus, you must also update the AWS-provided components to newer versions. You can configure the version of these components when you review the deployment later in this tutorial. For more information, see Update the AWS IoT Greengrass Core software (OTA).

b. For the **aws.greengrass.clientdevices.Auth** component, choose **Edit configuration**.

c. In the **Edit configuration** modal for the client device auth component, configure an authorization policy that allows client devices to publish and subscribe to the MQTT broker on the core device. Do the following:

i. Under **Configuration**, in the **Configuration to merge** code block, enter the following configuration, which contains a *client device authorization policy*. Each device group authorization policy specifies a set of actions and the resources on which a client device can perform those actions.

- This policy allows client devices whose names start with `MyClientDevice` to connect and communicate on all MQTT topics. Replace *`MyClientDevice*`* with the name of the AWS IoT thing to connect as a client device. You can also specify a name with the `*` wildcard that matches the client device's name. The `*` wildcard must be at the end of the name.

  If you have a second client device to connect, replace *`MyOtherClientDevice*`* with the name of that client device, or a wildcard pattern that matches that client device's name. Otherwise, you can remove or keep this section of the selection rule that allows client devices with names that match `MyOtherClientDevice*` to connect and communicate.

- This policy uses an `OR` operator to also allow client devices whose names start with `MyOtherClientDevice` to connect and communicate on all MQTT topics. You can

remove this clause in the selection rule or modify it to match the client devices to connect.

- This policy allows the client devices to publish and subscribe on all MQTT topics. To follow best security practices, restrict the `mqtt:publish` and `mqtt:subscribe` operations to the minimal set of topics that the client devices use to communicate.

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyDeviceGroup": {
        "selectionRule": "thingName: MyClientDevice* OR
 thingName: MyOtherClientDevice*",
        "policyName": "MyClientDevicePolicy"
      }
    },
    "policies": {
      "MyClientDevicePolicy": {
        "AllowConnect": {
          "statementDescription": "Allow client devices to connect.",
          "operations": [
            "mqtt:connect"
          ],
          "resources": [
            "*"
          ]
        },
        "AllowPublish": {
          "statementDescription": "Allow client devices to publish to all
 topics.",
          "operations": [
            "mqtt:publish"
          ],
          "resources": [
            "*"
          ]
        },
        "AllowSubscribe": {
          "statementDescription": "Allow client devices to subscribe to all
 topics.",
          "operations": [
```

```
            "mqtt:subscribe"
          ],
          "resources": [
            "*"
          ]
        }
      }
    }
  }
}
```

For more information, see Client device auth component configuration.

ii. Choose **Confirm**.

d. For the **aws.greengrass.clientdevices.mqtt.Bridge** component, choose **Edit configuration**.

e. In the **Edit configuration** modal for the MQTT bridge component, configure a topic mapping that relays MQTT messages from client devices to AWS IoT Core. Do the following:

i. Under **Configuration**, in the **Configuration to merge** code block, enter the following configuration. This configuration specifies to relay MQTT messages on the `clients/+/hello/world` topic filter from client devices to the AWS IoT Core cloud service. For example, this topic filter matches the `clients/MyClientDevice1/hello/world` topic.

```
{
  "mqttTopicMapping": {
    "HelloWorldIotCoreMapping": {
      "topic": "clients/+/hello/world",
      "source": "LocalMqtt",
      "target": "IotCore"
    }
  }
}
```

For more information, see MQTT bridge component configuration.

ii. Choose **Confirm**.

8. Choose **Review and deploy** to review the deployment that this page creates for you.

9. If you haven't previously set up the Greengrass service role in this Region, the console opens a modal to set up the service role for you. The client device auth component uses this service role to verify the identity of client devices, and the IP detector component uses this service role to manage core device connectivity information. Choose **Grant permissions**.
10. On the **Review** page, choose **Deploy** to start the deployment to the core device.
11. To verify that the deployment succeeds, check the status of the deployment, and check the logs on the core device. To check the status of the deployment on the core device, you can choose **Target** in the deployment **Overview**. For more information, see the following:

    - Check deployment status
    - Monitor AWS IoT Greengrass logs

Step 3: Connect client devices

Client devices can use the AWS IoT Device SDK to discover, connect, and communicate with a core device. The client device must be an AWS IoT thing. For more information, see Create a thing object in the *AWS IoT Core Developer Guide*.

In this section, you install the AWS IoT Device SDK v2 for Python and run the Greengrass discovery sample application from the AWS IoT Device SDK.

> **Note**
>
> The AWS IoT Device SDK is also available in other programming languages. This tutorial uses the AWS IoT Device SDK v2 for Python, but you can explore the other SDKs for your use case. For more information, see AWS IoT Device SDKs in the *AWS IoT Core Developer Guide*.

To connect a client device to a core device

1. Download and install the AWS IoT Device SDK v2 for Python to the AWS IoT thing to connect as a client device.

   On the client device, do the following:

   a. Clone the AWS IoT Device SDK v2 for Python repository to download it.

```
git clone https://github.com/aws/aws-iot-device-sdk-python-v2.git
```

b. Install the AWS IoT Device SDK v2 for Python.

```
python3 -m pip install --user ./aws-iot-device-sdk-python-v2
```

2. Change to the samples folder in the AWS IoT Device SDK v2 for Python.

```
cd aws-iot-device-sdk-python-v2/samples/greengrass
```

3. Run the sample Greengrass discovery application. This application expects arguments that specify the client device thing name, the MQTT topic and message to use, and the certificates that authenticate and secure the connection. The following example sends a Hello World message to the `clients/`*`MyClientDevice1`*`/hello/world` topic.

> **Note**
>
> This topic matches the topic where you configured the MQTT bridge to relay messages to AWS IoT Core earlier.

- Replace *`MyClientDevice1`* with the client device's thing name.
- Replace *`~/certs/AmazonRootCA1.pem`* with the path to the Amazon root CA certificate on the client device.
- Replace *`~/certs/device.pem.crt`* with the path to the device certificate on the client device.
- Replace *`~/certs/private.pem.key`* with the path to the private key file on the client device.
- Replace *`us-east-1`* with the AWS Region where your client device and core device operate.

```
python3 basic_discovery.py \\
  --thing_name MyClientDevice1 \\
  --topic 'clients/MyClientDevice1/hello/world' \\
  --message 'Hello World!' \\
  --ca_file ~/certs/AmazonRootCA1.pem \\
  --cert ~/certs/device.pem.crt \\
  --key ~/certs/private.pem.key \\
```

```
  --region us-east-1 \\
  --verbosity Warn
```

The discovery sample application sends the message 10 times and disconnects. It also subscribes to the same topic where it publishes messages. If the output indicates that the application received MQTT messages on the topic, the client device can successfully communicate with the core device.

```
Performing greengrass discovery...
awsiot.greengrass_discovery.DiscoverResponse(gg_groups=[awsiot.greengrass_discovery.GGGroup
coreDevice-MyGreengrassCore',
 cores=[awsiot.greengrass_discovery.GGCore(thing_arn='arn:aws:iot:us-
east-1:123456789012:thing/MyGreengrassCore',
 connectivity=[awsiot.greengrass_discovery.ConnectivityInfo(id='203.0.113.0',
 host_address='203.0.113.0', metadata='', port=8883)])],
 certificate_authorities=['-----BEGIN CERTIFICATE-----\
MIICiT...EXAMPLE=\
-----END CERTIFICATE-----\
'])])
Trying core arn:aws:iot:us-east-1:123456789012:thing/MyGreengrassCore at host
 203.0.113.0 port 8883
Connected!
Published topic clients/MyClientDevice1/hello/world: {"message": "Hello World!",
 "sequence": 0}

Publish received on topic clients/MyClientDevice1/hello/world
b'{"message": "Hello World!", "sequence": 0}'
Published topic clients/MyClientDevice1/hello/world: {"message": "Hello World!",
 "sequence": 1}

Publish received on topic clients/MyClientDevice1/hello/world
b'{"message": "Hello World!", "sequence": 1}'

...

Published topic clients/MyClientDevice1/hello/world: {"message": "Hello World!",
 "sequence": 9}

Publish received on topic clients/MyClientDevice1/hello/world
b'{"message": "Hello World!", "sequence": 9}'
```

If the application outputs an error instead, see Troubleshooting Greengrass discovery issues.

You can also view the Greengrass logs on the core device to verify if the client device successfully connects and sends messages. For more information, see Monitor AWS IoT Greengrass logs.

4. Verify that the MQTT bridge relays the messages from the client device to AWS IoT Core. You can use the MQTT test client in the AWS IoT Core console to subscribe to an MQTT topic filter. Do the following:

   a. Navigate to the AWS IoT console.

   b. In the left navigation menu, under **Test**, choose **MQTT test client**.

   c. On the **Subscribe to a topic** tab, for **Topic filter**, enter `clients/+/hello/world` to subscribe to client device messages from the core device.

   d. Choose **Subscribe**.

   e. Run the publish/subscribe application on the client device again.

      The MQTT test client displays the messages that you send from the client device on topics that match this topic filter.

Step 4: Develop a component that communicates with client devices

You can develop Greengrass components that communicate with client devices. Components use interprocess communication (IPC) and the local publish/subscribe interface to communicate on a core device. To interact with client devices, configure the MQTT bridge component to relay messages between client devices and the local publish/subscribe interface.

In this section, you update the MQTT bridge component to relay messages from client devices to the local publish/subscribe interface. Then, you develop a component that subscribes to these messages and prints the messages when it receives them.

To develop a component that communicates with client devices

1. Revise the deployment to the core device and configure the MQTT bridge component to relay messages from client devices to local publish/subscribe. Do the following:

   a. Navigate to the AWS IoT Greengrass console.

   b. In the left navigation menu, choose **Core devices**.

   c. On the **Core devices** page, choose the core device that you are using for this tutorial.

d. On the core device details page, choose the **Client devices** tab.

e. On the **Client devices** tab, choose **Configure cloud discovery**.

   The **Configure core device discovery** page opens. On this page, you can change or configure which client device components deploy to the core device.

f. In **Step 3**, for the **aws.greengrass.clientdevices.mqtt.Bridge** component, choose **Edit configuration**.

g. In the **Edit configuration** modal for the MQTT bridge component, configure a topic mapping that relays MQTT messages from client devices to the local publish/subscribe interface. Do the following:

   i. Under **Configuration**, in the **Configuration to merge** code block, enter the following configuration. This configuration specifies to relay MQTT messages on topics that match the `clients/+/hello/world` topic filter from client devices to the AWS IoT Core cloud service and the local Greengrass publish/subscribe broker.

```
{
  "mqttTopicMapping": {
    "HelloWorldIotCoreMapping": {
      "topic": "clients/+/hello/world",
      "source": "LocalMqtt",
      "target": "IotCore"
    },
    "HelloWorldPubsubMapping": {
      "topic": "clients/+/hello/world",
      "source": "LocalMqtt",
      "target": "Pubsub"
    }
  }
}
```

   For more information, see MQTT bridge component configuration.

   ii. Choose **Confirm**.

h. Choose **Review and deploy** to review the deployment that this page creates for you.

i. On the **Review** page, choose **Deploy** to start the deployment to the core device.

j. To verify that the deployment succeeds, check the status of the deployment, and check the logs on the core device. To check the status of the deployment on the core device, you can choose **Target** in the deployment **Overview**. For more information, see the following:

- Check deployment status
- Monitor AWS IoT Greengrass logs

2. Develop and deploy a Greengrass component that subscribes to Hello World messages from client devices. Do the following:

   a. Create folders for recipes and artifacts on the core device.

   Linux or Unix

   ```
   mkdir recipes
   mkdir -p artifacts/com.example.clientdevices.MyHelloWorldSubscriber/1.0.0
   ```

   Windows Command Prompt (CMD)

   ```
   mkdir recipes
   mkdir artifacts\com.example.clientdevices.MyHelloWorldSubscriber\1.0.0
   ```

   PowerShell

   ```
   mkdir recipes
   mkdir artifacts\com.example.clientdevices.MyHelloWorldSubscriber\1.0.0
   ```

   Important

   You must use the following format for the artifact folder path. Include the component name and version that you specify in the recipe.

   ```
   artifacts/componentName/componentVersion/
   ```

   b. Use a text editor to create a component recipe with the following contents. This recipe specifies to install the AWS IoT Device SDK v2 for Python and run a script that subscribes to the topic and prints messages.

   For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

```
nano recipes/com.example.clientdevices.MyHelloWorldSubscriber-1.0.0.json
```

Copy the following recipe into the file.

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.clientdevices.MyHelloWorldSubscriber",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that subscribes to Hello World messages
 from client devices.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
          "com.example.clientdevices.MyHelloWorldSubscriber:pubsub:1": {
            "policyDescription": "Allows access to subscribe to all topics.",
            "operations": [
              "aws.greengrass#SubscribeToTopic"
            ],
            "resources": [
              "*"
            ]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "install": "python3 -m pip install --user awsiotsdk",
        "Run": "python3 -u {artifacts:path}/hello_world_subscriber.py"
      }
    },
    {
      "Platform": {
        "os": "windows"
      },
```

```
      "Lifecycle": {
        "install": "py -3 -m pip install --user awsiotsdk",
        "Run": "py -3 -u {artifacts:path}/hello_world_subscriber.py"
      }
    }
  ]
}
```

c. Use a text editor to create a Python script artifact named `hello_world_subscriber.py` with the following contents. This application uses the publish/subscribe IPC service to subscribe to the `clients/+/hello/world` topic and print messages that it receives.

   For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

```
nano artifacts/com.example.clientdevices.MyHelloWorldSubscriber/1.0.0/
hello_world_subscriber.py
```

   Copy the following Python code into the file.

```
import sys
import time
import traceback

from awsiot.greengrasscoreipc.clientv2 import GreengrassCoreIPCClientV2

CLIENT_DEVICE_HELLO_WORLD_TOPIC = 'clients/+/hello/world'
TIMEOUT = 10


def on_hello_world_message(event):
    try:
        message = str(event.binary_message.message, 'utf-8')
        print('Received new message: %s' % message)
    except:
        traceback.print_exc()


try:
    ipc_client = GreengrassCoreIPCClientV2()
```

```
    # SubscribeToTopic returns a tuple with the response and the operation.
    _, operation = ipc_client.subscribe_to_topic(
        topic=CLIENT_DEVICE_HELLO_WORLD_TOPIC,
 on_stream_event=on_hello_world_message)
    print('Successfully subscribed to topic: %s' %
          CLIENT_DEVICE_HELLO_WORLD_TOPIC)

    # Keep the main thread alive, or the process will exit.
    try:
        while True:
            time.sleep(10)
    except InterruptedError:
        print('Subscribe interrupted.')

    operation.close()
except Exception:
    print('Exception occurred when using IPC.', file=sys.stderr)
    traceback.print_exc()
    exit(1)
```

> **Note**
>
> This component uses the IPC client V2 in the AWS IoT Device SDK v2 for Python to communicate with the AWS IoT Greengrass Core software. Compared to the original IPC client, the IPC client V2 reduces the amount of code that you need to write to use IPC in custom components.

d. Use the Greengrass CLI to deploy the component.

Linux or Unix

```
sudo /greengrass/v2/bin/greengrass-cli deployment create \
  --recipeDir recipes \
  --artifactDir artifacts \
  --merge "com.example.clientdevices.MyHelloWorldSubscriber=1.0.0"
```

Windows Command Prompt (CMD)

```
C:\greengrass\v2/bin/greengrass-cli deployment create ^
  --recipeDir recipes ^
  --artifactDir artifacts ^
```

```
  --merge "com.example.clientdevices.MyHelloWorldSubscriber=1.0.0"
```

PowerShell

```
C:\greengrass\v2/bin/greengrass-cli deployment create `
  --recipeDir recipes `
  --artifactDir artifacts `
  --merge "com.example.clientdevices.MyHelloWorldSubscriber=1.0.0"
```

3. View the component logs to verify that the component installs successfully and subscribes to the topic.

   Linux or Unix

   ```
   sudo tail -f /greengrass/v2/logs/
   com.example.clientdevices.MyHelloWorldSubscriber.log
   ```

   PowerShell

   ```
   gc C:\greengrass\v2/logs/com.example.clientdevices.MyHelloWorldSubscriber.log -
   Tail 10 -Wait
   ```

   You can keep the log feed open to verify that the core device receives messages.

4. On the client device, run the sample Greengrass discovery application again to send messages to the core device.

   ```
   python3 basic_discovery.py \\
     --thing_name MyClientDevice1 \\
     --topic 'clients/MyClientDevice1/hello/world' \\
     --message 'Hello World!' \\
     --ca_file ~/certs/AmazonRootCA1.pem \\
     --cert ~/certs/device.pem.crt \\
     --key ~/certs/private.pem.key \\
     --region us-east-1 \\
     --verbosity Warn
   ```

5. View the component logs again to verify that the component receives and prints the messages from the client device.

Linux or Unix

```
sudo tail -f /greengrass/v2/logs/
com.example.clientdevices.MyHelloWorldSubscriber.log
```

PowerShell

```
gc C:\greengrass\v2/logs/com.example.clientdevices.MyHelloWorldSubscriber.log -
Tail 10 -Wait
```

Step 5: Develop a component that interacts with client device shadows

You can develop Greengrass components that interact with client device's AWS IoT device shadows. A *shadow* is a JSON document that stores the current or desired state information for an AWS IoT thing, such as a client device. Custom components can access client devices' shadows to manage their state, even when the client device isn't connected to AWS IoT. Each AWS IoT thing has an unnamed shadow, and you can also create multiple named shadows for each thing.

In this section, you deploy the shadow manager component to manage shadows on the core device. You also update the MQTT bridge component to relay shadow messages between client devices and the shadow manager component. Then, you develop a component that updates the client devices' shadows, and you run a sample application on the client devices that responds to shadow updates from the component. This component represents a smart light management application, where the core device manages the color state of smart lights that connect to it as client devices.

To develop a component that interacts with client device shadows

1. Revise the deployment to the core device to deploy the shadow manager component and configure the MQTT bridge component to relay shadow messages between client devices and local publish/subscribe, where shadow manager communicates. Do the following:

   a. Navigate to the AWS IoT Greengrass console.

   b. In the left navigation menu, choose **Core devices**.

   c. On the **Core devices** page, choose the core device that you are using for this tutorial.

   d. On the core device details page, choose the **Client devices** tab.

e. On the **Client devices** tab, choose **Configure cloud discovery**.

   The **Configure core device discovery** page opens. On this page, you can change or configure which client device components deploy to the core device.

f. In **Step 3**, for the **aws.greengrass.clientdevices.mqtt.Bridge** component, choose **Edit configuration**.

g. In the **Edit configuration** modal for the MQTT bridge component, configure a topic mapping that relays MQTT messages on device shadow topics between client devices and the local publish/subscribe interface. You also confirm that the deployment specifies a compatible MQTT bridge version. Client device shadow support requires MQTT bridge v2.2.0 or later. Do the following:

   i. For **Component version**, choose version 2.2.0 or later.

   ii. Under **Configuration**, in the **Configuration to merge** code block, enter the following configuration. This configuration specifies to relays MQTT messages on shadow topics.

```
{
  "mqttTopicMapping": {
    "HelloWorldIotCoreMapping": {
      "topic": "clients/+/hello/world",
      "source": "LocalMqtt",
      "target": "IotCore"
    },
    "HelloWorldPubsubMapping": {
      "topic": "clients/+/hello/world",
      "source": "LocalMqtt",
      "target": "Pubsub"
    },
    "ShadowsLocalMqttToPubsub": {
      "topic": "$aws/things/+/shadow/#",
      "source": "LocalMqtt",
      "target": "Pubsub"
    },
    "ShadowsPubsubToLocalMqtt": {
      "topic": "$aws/things/+/shadow/#",
      "source": "Pubsub",
      "target": "LocalMqtt"
    }
  }
```

```
}
```

For more information, see MQTT bridge component configuration.

iii. Choose **Confirm**.

h. In **Step 3**, select the **aws.greengrass.ShadowManager** component to deploy it.

i. Choose **Review and deploy** to review the deployment that this page creates for you.

j. On the **Review** page, choose **Deploy** to start the deployment to the core device.

k. To verify that the deployment succeeds, check the status of the deployment, and check the logs on the core device. To check the status of the deployment on the core device, you can choose **Target** in the deployment **Overview**. For more information, see the following:

- Check deployment status
- Monitor AWS IoT Greengrass logs

2. Develop and deploy a Greengrass component that manages smart light client devices. Do the following:

a. Create a folder the component's artifacts on the core device.

Linux or Unix

```
mkdir -p artifacts/com.example.clientdevices.MySmartLightManager/1.0.0
```

Windows Command Prompt (CMD)

```
mkdir artifacts\com.example.clientdevices.MySmartLightManager\1.0.0
```

PowerShell

```
mkdir artifacts\com.example.clientdevices.MySmartLightManager\1.0.0
```

> **Important**
>
> You must use the following format for the artifact folder path. Include the component name and version that you specify in the recipe.

```
artifacts/componentName/componentVersion/
```

b. Use a text editor to create a component recipe with the following contents. This recipe specifies to install the AWS IoT Device SDK v2 for Python and run a script that interacts with smart light client devices' shadows to manage their colors.

For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

```
nano recipes/com.example.clientdevices.MySmartLightManager-1.0.0.json
```

Copy the following recipe into the file.

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.clientdevices.MySmartLightManager",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that interacts with smart light client
 devices.",
  "ComponentPublisher": "Amazon",
  "ComponentDependencies": {
    "aws.greengrass.Nucleus": {
      "VersionRequirement": "^2.6.0"
    },
    "aws.greengrass.ShadowManager": {
      "VersionRequirement": "^2.2.0"
    },
    "aws.greengrass.clientdevices.mqtt.Bridge": {
      "VersionRequirement": "^2.2.0"
    }
  },
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "smartLightDeviceNames": [],
      "accessControl": {
        "aws.greengrass.ShadowManager": {
          "com.example.clientdevices.MySmartLightManager:shadow:1": {
            "policyDescription": "Allows access to client devices' unnamed
 shadows",
            "operations": [
```

```
              "aws.greengrass#GetThingShadow",
              "aws.greengrass#UpdateThingShadow"
            ],
            "resources": [
              "$aws/things/MyClientDevice*/shadow"
            ]
          }
        },
        "aws.greengrass.ipc.pubsub": {
          "com.example.clientdevices.MySmartLightManager:pubsub:1": {
            "policyDescription": "Allows access to client devices' unnamed
shadow updates",
            "operations": [
              "aws.greengrass#SubscribeToTopic"
            ],
            "resources": [
              "$aws/things/+/shadow/update/accepted"
            ]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "install": "python3 -m pip install --user awsiotsdk",
        "Run": "python3 -u {artifacts:path}/smart_light_manager.py"
      }
    },
    {
      "Platform": {
        "os": "windows"
      },
      "Lifecycle": {
        "install": "py -3 -m pip install --user awsiotsdk",
        "Run": "py -3 -u {artifacts:path}/smart_light_manager.py"
      }
    }
  ]
```

```
}
```

c. Use a text editor to create a Python script artifact named `smart_light_manager.py` with the following contents. This application uses the shadow IPC service to get and update client device shadows and the local publish/subscribe IPC service to receive reported shadow updates.

For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

```
nano artifacts/com.example.clientdevices.MySmartLightManager/1.0.0/
smart_light_manager.py
```

Copy the following Python code into the file.

```
import json
import random
import sys
import time
import traceback
from uuid import uuid4

from awsiot.greengrasscoreipc.clientv2 import GreengrassCoreIPCClientV2
from awsiot.greengrasscoreipc.model import ResourceNotFoundError

SHADOW_COLOR_PROPERTY = 'color'
CONFIGURATION_CLIENT_DEVICE_NAMES = 'smartLightDeviceNames'
COLORS = ['red', 'orange', 'yellow', 'green', 'blue', 'purple']
SHADOW_UPDATE_TOPIC = '$aws/things/+/shadow/update/accepted'
SET_COLOR_INTERVAL = 15


class SmartLightDevice():
    def __init__(self, client_device_name: str, reported_color: str = None):
        self.name = client_device_name
        self.reported_color = reported_color
        self.desired_color = None


class SmartLightDeviceManager():
    def __init__(self, ipc_client: GreengrassCoreIPCClientV2):
        self.ipc_client = ipc_client
```

```
        self.devices = {}
        self.client_tokens = set()
        self.shadow_update_accepted_subscription_operation = None
        self.client_device_names_configuration_subscription_operation = None
        self.update_smart_light_device_list()

    def update_smart_light_device_list(self):
        # Update the device list from the component configuration.
        response = self.ipc_client.get_configuration(
            key_path=[CONFIGURATION_CLIENT_DEVICE_NAMES])
        # Identify the difference between the configuration and the currently
tracked devices.
        current_device_names = self.devices.keys()
        updated_device_names =
response.value[CONFIGURATION_CLIENT_DEVICE_NAMES]
        added_device_names = set(updated_device_names) -
set(current_device_names)
        removed_device_names = set(current_device_names) -
set(updated_device_names)
        # Stop tracking any smart light devices that are no longer in the
configuration.
        for name in removed_device_names:
            print('Removing %s from smart light device manager' % name)
            self.devices.pop(name)
        # Start tracking any new smart light devices that are in the
configuration.
        for name in added_device_names:
            print('Adding %s to smart light device manager' % name)
            device = SmartLightDevice(name)
            device.reported_color = self.get_device_reported_color(device)
            self.devices[name] = device
            print('Current color for %s is %s' % (name,
device.reported_color))

    def get_device_reported_color(self, smart_light_device):
        try:
            response = self.ipc_client.get_thing_shadow(
                thing_name=smart_light_device.name, shadow_name='')
            shadow = json.loads(str(response.payload, 'utf-8'))
            if 'reported' in shadow['state']:
                return shadow['state']['reported'].get(SHADOW_COLOR_PROPERTY)
            return None
        except ResourceNotFoundError:
            return None
```

```
    def request_device_color_change(self, smart_light_device, color):
        # Generate and track a client token for the request.
        client_token = str(uuid4())
        self.client_tokens.add(client_token)
        # Create a shadow payload, which must be a blob.
        payload_json = {
            'state': {
                'desired': {
                    SHADOW_COLOR_PROPERTY: color
                }
            },
            'clientToken': client_token
        }
        payload = bytes(json.dumps(payload_json), 'utf-8')
        self.ipc_client.update_thing_shadow(
            thing_name=smart_light_device.name, shadow_name='',
payload=payload)
        smart_light_device.desired_color = color

    def subscribe_to_shadow_update_accepted_events(self):
        if self.shadow_update_accepted_subscription_operation == None:
            # SubscribeToTopic returns a tuple with the response and the
operation.
            _, self.shadow_update_accepted_subscription_operation =
self.ipc_client.subscribe_to_topic(
                topic=SHADOW_UPDATE_TOPIC,
on_stream_event=self.on_shadow_update_accepted_event)
            print('Successfully subscribed to shadow update accepted topic')

    def close_shadow_update_accepted_subscription(self):
        if self.shadow_update_accepted_subscription_operation is not None:
            self.shadow_update_accepted_subscription_operation.close()

    def on_shadow_update_accepted_event(self, event):
        try:
            message = str(event.binary_message.message, 'utf-8')
            accepted_payload = json.loads(message)
            # Check for reported states from smart light devices and ignore
desired states from components.
            if 'reported' in accepted_payload['state']:
                # Process this update only if it uses a client token created by
this component.
                client_token = accepted_payload.get('clientToken')
```

```
                if client_token is not None and client_token in
self.client_tokens:
                    self.client_tokens.remove(client_token)
                    shadow_state = accepted_payload['state']['reported']
                    if SHADOW_COLOR_PROPERTY in shadow_state:
                        reported_color = shadow_state[SHADOW_COLOR_PROPERTY]
                        topic = event.binary_message.context.topic
                        client_device_name = topic.split('/')[2]
                        if client_device_name in self.devices:
                            # Set the reported color for the smart light
device.
                            self.devices[client_device_name].reported_color =
reported_color
                            print(
                                'Received shadow update confirmation from
client device: %s' % client_device_name)
                    else:
                        print("Shadow update doesn't specify color")
        except:
            traceback.print_exc()

    def subscribe_to_client_device_name_configuration_updates(self):
        if self.client_device_names_configuration_subscription_operation ==
None:
            # SubscribeToConfigurationUpdate returns a tuple with the response
and the operation.
            _, self.client_device_names_configuration_subscription_operation =
self.ipc_client.subscribe_to_configuration_update(
                key_path=[CONFIGURATION_CLIENT_DEVICE_NAMES],
on_stream_event=self.on_client_device_names_configuration_update_event)
            print(
                'Successfully subscribed to configuration updates for smart
light device names')

    def close_client_device_names_configuration_subscription(self):
        if self.client_device_names_configuration_subscription_operation is not
None:

self.client_device_names_configuration_subscription_operation.close()

    def on_client_device_names_configuration_update_event(self, event):
        try:
            if CONFIGURATION_CLIENT_DEVICE_NAMES in
event.configuration_update_event.key_path:
```

```
                print('Received configuration update for list of client
 devices')
                self.update_smart_light_device_list()
        except:
            traceback.print_exc()


def choose_random_color():
    return random.choice(COLORS)

def main():
    try:
        # Create an IPC client and a smart light device manager.
        ipc_client = GreengrassCoreIPCClientV2()
        smart_light_manager = SmartLightDeviceManager(ipc_client)
        smart_light_manager.subscribe_to_shadow_update_accepted_events()

 smart_light_manager.subscribe_to_client_device_name_configuration_updates()
        try:
            # Keep the main thread alive, or the process will exit.
            while True:
                # Set each smart light device to a random color at a regular
 interval.
                for device_name in smart_light_manager.devices:
                    device = smart_light_manager.devices[device_name]
                    desired_color = choose_random_color()
                    print('Chose random color (%s) for %s' %
                          (desired_color, device_name))
                    if desired_color == device.desired_color:
                        print('Desired color for %s is already %s' %
                              (device_name, desired_color))
                    elif desired_color == device.reported_color:
                        print('Reported color for %s is already %s' %
                              (device_name, desired_color))
                    else:
                        smart_light_manager.request_device_color_change(
                            device, desired_color)
                        print('Requested color change for %s to %s' %
                              (device_name, desired_color))
                time.sleep(SET_COLOR_INTERVAL)
        except InterruptedError:
            print('Application interrupted')
        smart_light_manager.close_shadow_update_accepted_subscription()
```

```
 smart_light_manager.close_client_device_names_configuration_subscription()
    except Exception:
        print('Exception occurred', file=sys.stderr)
        traceback.print_exc()
        exit(1)


if __name__ == '__main__':
    main()
```

This Python application does the following:

- Reads the component's configuration to get the list of smart light client devices to manage.
- Subscribes to configuration update notifications using the SubscribeToConfigurationUpdate IPC operation. The AWS IoT Greengrass Core software sends notifications each time the component's configuration changes. When the component receives a configuration update notification, it updates the list of smart light client devices that it manages.
- Gets each smart light client device's shadow to get its initial color state.
- Sets each smart light client device's color to a random color every 15 seconds. The component updates the client device's thing shadow to change its color. This operation sends a shadow delta event to the client device over MQTT.
- Subscribes to shadow update accepted messages on the local publish/subscribe interface using the SubscribeToTopic IPC operation. This component receives these messages to track the color of each smart light client device. When a smart light client device receives a shadow update, it sends an MQTT message to confirm that it received the update. The MQTT bridge relays this message to the local publish/subscribe interface.

d. Use the Greengrass CLI to deploy the component. When you deploy this component, you specify the list of client devices, `smartLightDeviceNames`, whose shadows it manages. Replace *`MyClientDevice1`* with the client device's thing name.

   Linux or Unix

   ```
   sudo /greengrass/v2/bin/greengrass-cli deployment create \
     --recipeDir recipes \
   ```

```
  --artifactDir artifacts \
  --merge "com.example.clientdevices.MySmartLightManager=1.0.0" \
  --update-config '{
    "com.example.clientdevices.MySmartLightManager": {
      "MERGE": {
        "smartLightDeviceNames": [
          "MyClientDevice1"
        ]
      }
    }
  }'
```

Windows Command Prompt (CMD)

```
C:\greengrass\v2/bin/greengrass-cli deployment create ^
  --recipeDir recipes ^
  --artifactDir artifacts ^
  --merge "com.example.clientdevices.MySmartLightManager=1.0.0" ^
  --update-config '{"com.example.clientdevices.MySmartLightManager":
{"MERGE":{"smartLightDeviceNames":["MyClientDevice1"]}}}'
```

PowerShell

```
C:\greengrass\v2/bin/greengrass-cli deployment create `
  --recipeDir recipes `
  --artifactDir artifacts `
  --merge "com.example.clientdevices.MySmartLightManager=1.0.0" `
  --update-config '{
    "com.example.clientdevices.MySmartLightManager": {
      "MERGE": {
        "smartLightDeviceNames": [
          "MyClientDevice1"
        ]
      }
    }
  }'
```

3. View the component logs to verify that the component installs and runs successfully.

Linux or Unix

```
sudo tail -f /greengrass/v2/logs/
com.example.clientdevices.MySmartLightManager.log
```

PowerShell

```
gc C:\greengrass\v2/logs/com.example.clientdevices.MySmartLightManager.log -Tail
 10 -Wait
```

The component sends requests to change the color of the smart light client device. The shadow manager receives the request and sets the shadow's `desired` state. However, the smart light client device isn't running yet, so the shadow's `reported` state doesn't change. The component's logs include the following messages.

```
2022-07-07T03:49:24.908Z [INFO] (Copier)
 com.example.clientdevices.MySmartLightManager: stdout. Chose random color (blue)
 for MyClientDevice1.
 {scriptName=services.com.example.clientdevices.MySmartLightManager.lifecycle.Run,
 serviceName=com.example.clientdevices.MySmartLightManager, currentState=RUNNING}
2022-07-07T03:49:24.912Z [INFO] (Copier)
 com.example.clientdevices.MySmartLightManager: stdout.
 Requested color change for MyClientDevice1 to blue.
 {scriptName=services.com.example.clientdevices.MySmartLightManager.lifecycle.Run,
 serviceName=com.example.clientdevices.MySmartLightManager, currentState=RUNNING}
```

You can keep the log feed open to see when the component prints messages.

4. Download and run a sample application that uses Greengrass discovery and subscribes to device shadow updates. On the client device, do the following:

   a. Use a text editor to create a Python script named `basic_discovery_shadow.py` with the following contents. This application uses Greengrass discovery and shadows to keep a property in sync between the client device and the core device.

      For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

```
nano basic_discovery_shadow.py
```

Copy the following Python code into the file.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0.

from awscrt import io
from awscrt import mqtt
from awsiot import iotshadow
from awsiot.greengrass_discovery import DiscoveryClient
from awsiot import mqtt_connection_builder
from concurrent.futures import Future
import sys
import threading
import traceback
from uuid import uuid4

# ------------------------------- ARGUMENT PARSING
 ---------------------------------------
import argparse

def parse_sample_input():
    parser = argparse.ArgumentParser(
        description="AWS IoT Greengrass Discovery Shadow",
        formatter_class=argparse.ArgumentDefaultsHelpFormatter,
    )

    # Connection / TLS
    parser.add_argument("--cert", required=True, dest="input_cert",
                        help="Path to the certificate file to use during mTLS
 connection establishment")
    parser.add_argument("--key", required=True, dest="input_key",
                        help="Path to the private key file to use during mTLS
 connection establishment")
    parser.add_argument("--ca_file", dest="input_ca", help="Path to optional CA
 bundle (PEM)")

    # Shadow
    parser.add_argument("--thing_name", required=True, dest="input_thing_name",
 help="The name assigned to your IoT Thing.")
```

```
    parser.add_argument("--shadow_property", dest="input_shadow_property",
 default="color",
                        help="The name of the shadow property you want to
 change (optional, default='color'")
    parser.add_argument("--region", dest="input_region", help="The region to
 connect through.", required=True)
    parser.add_argument("--print_discover_resp_only",
 dest="input_print_discover_resp_only", default=False)

    return parser.parse_args()

args = parse_sample_input()

# ------------------------------- ARGUMENT PARSING END
 ---------------------------------------


# Using globals to simplify sample code
is_sample_done = threading.Event()
mqtt_connection = None
shadow_thing_name = args.input_thing_name
shadow_property = args.input_shadow_property

SHADOW_VALUE_DEFAULT = "off"

class LockedData:
    def __init__(self):
        self.lock = threading.Lock()
        self.shadow_value = None
        self.disconnect_called = False
        self.request_tokens = set()

locked_data = LockedData()


def on_connection_interupted(connection, error, **kwargs):
    print('connection interrupted with error {}'.format(error))


def on_connection_resumed(connection, return_code, session_present, **kwargs):
    print('connection resumed with return code {}, session present
 {}'.format(return_code, session_present))

# Try IoT endpoints until we find one that works
```

```
def try_iot_endpoints():
    for gg_group in discover_response.gg_groups:
        for gg_core in gg_group.cores:
            for connectivity_info in gg_core.connectivity:
                try:
                    print('Trying core {} at host {} port
 {}'.format(gg_core.thing_arn, connectivity_info.host_address,
 connectivity_info.port))
                    mqtt_connection = mqtt_connection_builder.mtls_from_path(
                        endpoint=connectivity_info.host_address,
                        port=connectivity_info.port,
                        cert_filepath=args.input_cert,
                        pri_key_filepath=args.input_key,

 ca_bytes=gg_group.certificate_authorities[0].encode('utf-8'),
                        on_connection_interrupted=on_connection_interupted,
                        on_connection_resumed=on_connection_resumed,
                        client_id=args.input_thing_name,
                        clean_session=False,
                        keep_alive_secs=30)

                    connect_future = mqtt_connection.connect()
                    connect_future.result()
                    print('Connected!')
                    return mqtt_connection

                except Exception as e:
                    print('Connection failed with exception {}'.format(e))
                    continue

    exit('All connection attempts failed')

# Function for gracefully quitting this sample
def exit(msg_or_exception):
    if isinstance(msg_or_exception, Exception):
        print("Exiting sample due to exception.")
        traceback.print_exception(msg_or_exception.__class__, msg_or_exception,
 sys.exc_info()[2])
    else:
        print("Exiting sample:", msg_or_exception)

    with locked_data.lock:
        if not locked_data.disconnect_called:
            print("Disconnecting...")
```

```
            locked_data.disconnect_called = True
            future = mqtt_connection.disconnect()
            future.add_done_callback(on_disconnected)

def on_disconnected(disconnect_future):
    # type: (Future) -> None
    print("Disconnected.")

    # Signal that sample is finished
    is_sample_done.set()

def on_get_shadow_accepted(response):
    # type: (iotshadow.GetShadowResponse) -> None
    try:
        with locked_data.lock:
            # check that this is a response to a request from this session
            try:
                locked_data.request_tokens.remove(response.client_token)
            except KeyError:
                return

            print("Finished getting initial shadow state.")
            if locked_data.shadow_value is not None:
                print("  Ignoring initial query because a delta event has
 already been received.")
                return

        if response.state:
            if response.state.delta:
                value = response.state.delta.get(shadow_property)
                if value:
                    print("  Shadow contains delta value '{}'.".format(value))
                    change_shadow_value(value)
                    return

            if response.state.reported:
                value = response.state.reported.get(shadow_property)
                if value:
                    print("  Shadow contains reported value
 '{}'.".format(value))

 set_local_value_due_to_initial_query(response.state.reported[shadow_property])
                    return
```

```
        print("  Shadow document lacks '{}' property. Setting
 defaults...".format(shadow_property))
        change_shadow_value(SHADOW_VALUE_DEFAULT)
        return

    except Exception as e:
        exit(e)

def on_get_shadow_rejected(error):
    # type: (iotshadow.ErrorResponse) -> None
    try:
        # check that this is a response to a request from this session
        with locked_data.lock:
            try:
                locked_data.request_tokens.remove(error.client_token)
            except KeyError:
                return

        if error.code == 404:
            print("Thing has no shadow document. Creating with defaults...")
            change_shadow_value(SHADOW_VALUE_DEFAULT)
        else:
            exit("Get request was rejected. code:{} message:'{}'".format(
                error.code, error.message))

    except Exception as e:
        exit(e)

def on_shadow_delta_updated(delta):
    # type: (iotshadow.ShadowDeltaUpdatedEvent) -> None
    try:
        print("Received shadow delta event.")
        if delta.state and (shadow_property in delta.state):
            value = delta.state[shadow_property]
            if value is None:
                print("  Delta reports that '{}' was deleted. Resetting
 defaults...".format(shadow_property))
                change_shadow_value(SHADOW_VALUE_DEFAULT)
                return
            else:
                print("  Delta reports that desired value is '{}'. Changing
 local value...".format(value))
                if (delta.client_token is not None):
                    print ("  ClientToken is: " + delta.client_token)
```

```
                change_shadow_value(value, delta.client_token)
        else:
            print("  Delta did not report a change in
 '{}'".format(shadow_property))

    except Exception as e:
        exit(e)

def on_publish_update_shadow(future):
    #type: (Future) -> None
    try:
        future.result()
        print("Update request published.")
    except Exception as e:
        print("Failed to publish update request.")
        exit(e)

def on_update_shadow_accepted(response):
    # type: (iotshadow.UpdateShadowResponse) -> None
    try:
        # check that this is a response to a request from this session
        with locked_data.lock:
            try:
                locked_data.request_tokens.remove(response.client_token)
            except KeyError:
                return

        try:
            if response.state.reported != None:
                if shadow_property in response.state.reported:
                    print("Finished updating reported shadow value to
 '{}'.".format(response.state.reported[shadow_property])) # type: ignore
                else:
                    print ("Could not find shadow property with name:
 '{}'.".format(shadow_property)) # type: ignore
            else:
                print("Shadow states cleared.") # when the shadow states are
 cleared, reported and desired are set to None
        except:
            exit("Updated shadow is missing the target property")

    except Exception as e:
        exit(e)
```

```
def on_update_shadow_rejected(error):
    # type: (iotshadow.ErrorResponse) -> None
    try:
        # check that this is a response to a request from this session
        with locked_data.lock:
            try:
                locked_data.request_tokens.remove(error.client_token)
            except KeyError:
                return

        exit("Update request was rejected. code:{} message:'{}'".format(
            error.code, error.message))

    except Exception as e:
        exit(e)

def set_local_value_due_to_initial_query(reported_value):
    with locked_data.lock:
        locked_data.shadow_value = reported_value

def change_shadow_value(value, token=None):
    with locked_data.lock:
        if locked_data.shadow_value == value:
            print("Local value is already '{}'.".format(value))
            return

        print("Changed local shadow value to '{}'.".format(value))
        locked_data.shadow_value = value

        print("Updating reported shadow value to '{}'...".format(value))

        reuse_token = token is not None
        # use a unique token so we can correlate this "request" message to
        # any "response" messages received on the /accepted and /rejected
 topics
        if not reuse_token:
            token = str(uuid4())

        # if the value is "clear shadow" then send a UpdateShadowRequest with
 None
        # for both reported and desired to clear the shadow document
 completely.
        if value == "clear_shadow":
```

```
            tmp_state = iotshadow.ShadowState(reported=None, desired=None,
 reported_is_nullable=True, desired_is_nullable=True)
            request = iotshadow.UpdateShadowRequest(
                thing_name=shadow_thing_name,
                state=tmp_state,
                client_token=token,
            )
        # Otherwise, send a normal update request
        else:
            # if the value is "none" then set it to a Python none object to
            # clear the individual shadow property
            if value == "none":
                value = None

            request = iotshadow.UpdateShadowRequest(
            thing_name=shadow_thing_name,
            state=iotshadow.ShadowState(
                reported={ shadow_property: value }
                ),
                client_token=token,
            )

        future = shadow_client.publish_update_shadow(request,
 mqtt.QoS.AT_LEAST_ONCE)

        if not reuse_token:
            locked_data.request_tokens.add(token)

        future.add_done_callback(on_publish_update_shadow)


if __name__ == '__main__':
    tls_options =
 io.TlsContextOptions.create_client_with_mtls_from_path(args.input_cert,args.input_key)
    if args.input_ca:
        tls_options.override_default_trust_store_from_path(None,
 args.input_ca)
    tls_context = io.ClientTlsContext(tls_options)

    socket_options = io.SocketOptions()

    print('Performing greengrass discovery...')
```

```
   discovery_client =
DiscoveryClient(io.ClientBootstrap.get_or_create_static_default(),
socket_options, tls_context, args.input_region)
   resp_future = discovery_client.discover(args.input_thing_name)
   discover_response = resp_future.result()

   print(discover_response)
   if args.print_discover_resp_only:
       exit(0)

   mqtt_connection = try_iot_endpoints()
   shadow_client = iotshadow.IotShadowClient(mqtt_connection)

   try:
       # Subscribe to necessary topics.
       # Note that is **is** important to wait for "accepted/rejected"
subscriptions
       # to succeed before publishing the corresponding "request".
       print("Subscribing to Update responses...")
       update_accepted_subscribed_future, _ =
shadow_client.subscribe_to_update_shadow_accepted(

request=iotshadow.UpdateShadowSubscriptionRequest(thing_name=shadow_thing_name),
           qos=mqtt.QoS.AT_LEAST_ONCE,
           callback=on_update_shadow_accepted)

       update_rejected_subscribed_future, _ =
shadow_client.subscribe_to_update_shadow_rejected(

request=iotshadow.UpdateShadowSubscriptionRequest(thing_name=shadow_thing_name),
           qos=mqtt.QoS.AT_LEAST_ONCE,
           callback=on_update_shadow_rejected)

       # Wait for subscriptions to succeed
       update_accepted_subscribed_future.result()
       update_rejected_subscribed_future.result()

       print("Subscribing to Get responses...")
       get_accepted_subscribed_future, _ =
shadow_client.subscribe_to_get_shadow_accepted(

request=iotshadow.GetShadowSubscriptionRequest(thing_name=shadow_thing_name),
           qos=mqtt.QoS.AT_LEAST_ONCE,
           callback=on_get_shadow_accepted)
```

```
        get_rejected_subscribed_future, _ =
shadow_client.subscribe_to_get_shadow_rejected(

request=iotshadow.GetShadowSubscriptionRequest(thing_name=shadow_thing_name),
            qos=mqtt.QoS.AT_LEAST_ONCE,
            callback=on_get_shadow_rejected)

        # Wait for subscriptions to succeed
        get_accepted_subscribed_future.result()
        get_rejected_subscribed_future.result()

        print("Subscribing to Delta events...")
        delta_subscribed_future, _ =
shadow_client.subscribe_to_shadow_delta_updated_events(

request=iotshadow.ShadowDeltaUpdatedSubscriptionRequest(thing_name=shadow_thing_name),
            qos=mqtt.QoS.AT_LEAST_ONCE,
            callback=on_shadow_delta_updated)

        # Wait for subscription to succeed
        delta_subscribed_future.result()

        # The rest of the sample runs asynchronously.

        # Issue request for shadow's current state.
        # The response will be received by the on_get_accepted() callback
        print("Requesting current shadow state...")

        with locked_data.lock:
            # use a unique token so we can correlate this "request" message to
            # any "response" messages received on the /accepted and /rejected
topics
            token = str(uuid4())

            publish_get_future = shadow_client.publish_get_shadow(

request=iotshadow.GetShadowRequest(thing_name=shadow_thing_name,
client_token=token),
                qos=mqtt.QoS.AT_LEAST_ONCE)

            locked_data.request_tokens.add(token)

        # Ensure that publish succeeds
```

```
        publish_get_future.result()

    except Exception as e:
        exit(e)

    # Wait for the sample to finish (user types 'quit', or an error occurs)
    is_sample_done.wait()
```

This Python application does the following:

- Uses Greengrass discovery to discover and connect to the core device.
- Requests the shadow document from the core device to get the property's initial state.
- Subscribes to shadow delta events, which the core device sends when the property's `desired` value differs from its `reported` value. When the application receives a shadow delta event, it changes the value of the property and sends an update to the core device to set the new value as its `reported` value.

This application combines the Greengrass discovery and shadow samples from the AWS IoT Device SDK v2.

b. Run the sample application. This application expects arguments that specify the client device thing name, the shadow property to use, and the certificates that authenticate and secure the connection.

- Replace *`MyClientDevice1`* with the client device's thing name.
- Replace *`~/certs/AmazonRootCA1.pem`* with the path to the Amazon root CA certificate on the client device.
- Replace *`~/certs/device.pem.crt`* with the path to the device certificate on the client device.
- Replace *`~/certs/private.pem.key`* with the path to the private key file on the client device.
- Replace *`us-east-1`* with the AWS Region where your client device and core device operate.

```
python3 basic_discovery_shadow.py \
  --thing_name MyClientDevice1 \
  --shadow_property color \
```

```
  --ca_file ~/certs/AmazonRootCA1.pem \
  --cert ~/certs/device.pem.crt \
  --key ~/certs/private.pem.key \
  --region us-east-1 \
  --verbosity Warn
```

The sample application subscribes to the shadow topics and waits to receive shadow delta events from the core device. If the output indicates that the application receives and responds to shadow delta events, the client device can successfully interact with its shadow on the core device.

```
Performing greengrass discovery...
awsiot.greengrass_discovery.DiscoverResponse(gg_groups=[awsiot.greengrass_discovery.GGG
coreDevice-MyGreengrassCore',
 cores=[awsiot.greengrass_discovery.GGCore(thing_arn='arn:aws:iot:us-
east-1:123456789012:thing/MyGreengrassCore',
 connectivity=[awsiot.greengrass_discovery.ConnectivityInfo(id='203.0.113.0',
 host_address='203.0.113.0', metadata='', port=8883)])],
 certificate_authorities=['-----BEGIN CERTIFICATE-----
\nMIICiT...EXAMPLE=\n-----END CERTIFICATE-----\n'])])
Trying core arn:aws:iot:us-east-1:123456789012:thing/MyGreengrassCore at host
 203.0.113.0 port 8883
Connected!
Subscribing to Update responses...
Subscribing to Get responses...
Subscribing to Delta events...
Requesting current shadow state...
Received shadow delta event.
  Delta reports that desired value is 'purple'. Changing local value...
  ClientToken is: 3dce4d3f-e336-41ac-aa4f-7882725f0033
Changed local shadow value to 'purple'.
Updating reported shadow value to 'purple'...
Update request published.
```

If the application outputs an error instead, see Troubleshooting Greengrass discovery issues.

You can also view the Greengrass logs on the core device to verify if the client device successfully connects and sends messages. For more information, see Monitor AWS IoT Greengrass logs.

5. View the component logs again to verify that the component receives shadow update confirmations from the smart light client device.

   Linux or Unix

   ```
   sudo tail -f /greengrass/v2/logs/
   com.example.clientdevices.MySmartLightManager.log
   ```

   PowerShell

   ```
   gc C:\greengrass\v2/logs/com.example.clientdevices.MySmartLightManager.log -Tail
    10 -Wait
   ```

   The component logs messages to confirm that smart light client device changed its color.

   ```
   2022-07-07T03:49:24.908Z [INFO] (Copier)
    com.example.clientdevices.MySmartLightManager: stdout. Chose random color (blue)
    for MyClientDevice1.
    {scriptName=services.com.example.clientdevices.MySmartLightManager.lifecycle.Run,
    serviceName=com.example.clientdevices.MySmartLightManager, currentState=RUNNING}
   2022-07-07T03:49:24.912Z [INFO] (Copier)
    com.example.clientdevices.MySmartLightManager: stdout.
    Requested color change for MyClientDevice1 to blue.
    {scriptName=services.com.example.clientdevices.MySmartLightManager.lifecycle.Run,
    serviceName=com.example.clientdevices.MySmartLightManager, currentState=RUNNING}
   2022-07-07T03:49:24.959Z [INFO] (Copier)
    com.example.clientdevices.MySmartLightManager: stdout. Received
    shadow update confirmation from client device: MyClientDevice1.
    {scriptName=services.com.example.clientdevices.MySmartLightManager.lifecycle.Run,
    serviceName=com.example.clientdevices.MySmartLightManager, currentState=RUNNING}
   ```

> **Note**
>
> The client device's shadow is in sync between the core device and the client device. However, the core device doesn't sync the client device's shadow with AWS IoT Core. You might sync a shadow with AWS IoT Core to view or modify the state of all devices in your fleet, for example. For more information about how to configure the shadow manager

component to sync shadows with AWS IoT Core, see Sync local device shadows with AWS IoT Core.

You've completed this tutorial. The client device connects to the core device, sends MQTT messages to AWS IoT Core and Greengrass components, and receives shadow updates from the core device. For more information about the topics covered in this tutorial, see the following:

- Associate client devices
- Manage core device endpoints
- Test client device communications
- Greengrass discovery RESTful API
- Relay MQTT messages between client devices and AWS IoT Core
- Interact with client devices in components
- Interact with device shadows
- Interact with and sync client device shadows

Tutorial: Secure Greengrass Nucleus with Trusted Platform Module (TPM)

Note

The mechanism in this tutorial is supported only with Install AWS IoT Greengrass Core software with manual resource provisioning.

These tutorials contain instructions on how to use the TPM2 chip as an Hardware Security Module(HSM) to create a private key and CSR. Which is used to Create the thing certificate.

These tutorials show you how to enhance device security by configuring AWS IoT Greengrass Core software (Greengrass Nucleus) with a Trusted Platform Module (TPM) using the PKCS#11 interface. This TPM integration ensures that private keys and certificates used for device identity and connecting to AWS IoT Core are securely stored in tamper-proof hardware, preventing extraction for impersonation or other malicious activities.

When you complete this integration, your Greengrass core device will use TPM-protected private keys for its identity and communication with AWS IoT services.

For more information about security on Greengrass devices, see Security in AWS IoT Greengrass.

Prerequisites

To complete this tutorial, you need the following:

- A Linux-compatible device with TPM 2.0 hardware or firmware TPM 2.0.
- Instructions in this tutorial are defined for Ubuntu 24.04 LTS.
  - Any Linux distribution that can support the Linux TPM2 Software Stack can support this mechanism.
- A developer machine with AWS CLI installed and configured with permissions to:
  - Create and manage AWS IoT resources
  - Create and manage IAM roles and policies
- Java Runtime Environment (JRE) version 8 or later installed on your device.
- The following software packages installed on your device:
  - curl
  - jq
- Root or sudo privileges on the device.

Step 1: Install TPM2 tools and dependencies

In this step, you install the necessary TPM2 software tools and libraries.

1. Update your package manager and install the TPM2 tools and dependencies by running the following command.

```
sudo apt-get update && sudo apt-get install tpm2-tools \
tpm2-abrmd \
tpm2-tss-engine-tools \
gnutls-bin \
libtpm2-pkcs11-1 \
libtpm2-pkcs11-tools \
```

```
libtpm2-pkcs11-1-dev \
python3-tpm2-pkcs11-tools \
libengine-pkcs11-openssl \
libtss2-tcti-tabrmd0
```

2. Install the OpenSSL TPM2 provider packages on Ubuntu 24.04 that use OpenSSL engine 3.

```
sudo apt-get install tpm2-openssl
```

Step 2: Initialize PKCS#11 store and create a slot

1. Create a directory to store data.

```
sudo mkdir -p /etc/tpm2_pkcs11
```

2. Set the storage location as an environment variable. For more information about the store hierarchy, see Initialization.

```
export TPM2_PKCS11_STORE=/etc/tpm2_store
```

3. Initialize the TPM2 token with the primary object.

```
sudo tpm2_ptool init
```

The options available are the following:

`hierarchy-auth HIERARCHY_AUTH`

The authorization password for adding a primary object to the hierarchy.

`primary-auth PRIMARY_AUTH`

Authorization value for existing primary key object.

Default is an empty auth value.

`primary-handle [PRIMARY_HANDLE]`

Use an existing primary key object.

Default: `0x81000001`

`transient-parents`

Use a transient primary object of a given template.

Values: `tpm2-tools-default`,`tpm2-tools-ecc-default`,`tss2-engine-key`

`path PATH`

The location of the store directory. If specified, the directory must exist. If not specified, it performs a search by looking at environment variable `TPM2_PKCS11_STORE`. If that environment variable is not set, it will look at `/etc/tpm2_pkcs11`. If that directory is not found or can't be created, it defaults to the current working directory.

Step 3: Create a token and a key

1. Create a PKCS#11 token.

```
sudo tpm2_ptool addtoken –pid=1 –userpin=USERPIN –sopin=SOPIN –label=greengrass
```

The available options are the following:

`--pid PID`

The primary object id to associate with this token.

`--sopin SOPIN`

The Administrator pin. This pin is used for object recovery.

`--userpin USERPIN`

The user pin. This pin is used for authentication for object use.

`--label LABEL`

A unique label to identify the profile in use, must be unique.

```
--hierarchy-auth HIERARCHY_AUTH
```

2. Create an ECC key object.

```
sudo tpm2_ptool addkey –algorithm=ecc256 –label=greengrass –userpin=****** –key-
label=greenkey
```

`--label LABEL`

The tokens label to import the key too.

`--key-label KEY_LABEL`

The label of the key imported. Defaults to an integer value.

`--id ID`

The key id. Defaults to a random 8 bytes of hex.

`--attr-always-authenticate`

Sets the CKA_ALWAYS_AUTHENTICATE attribute to CK_TRUE.

`--hierarchy-auth HIERARCHY_AUTH`

The hierarchyauth, required for transient pobjects.

`--sopin SOPIN`

The Administrator pin.

`--userpin USERPIN`

The User pin.

`--algorithm`

{rsa1024,rsa2048,rsa3072,rsa4096,aes128,aes256,ecc224,ecc256,ecc384,ecc521,hmac:sha1,hmac:sh

The type of the key.

3. Export the TPM2-TSS object from the token to capture the authentication data.

```
yaml_ecc0=$(sudo tpm2_ptool export –label="greengrass" –key-label="greenkey" –
userpin="******")
```

Example output:

```
> echo $yaml_ecc0
object-auth: 706c1cad8a5238871b30149705255926
primary-object:
  auth: ''
  hierarchy: owner
  is_transient: false
```

> **Note**
>
> You will also find a file named `greenkey.pem` which is the TSS2 Private Key in the directory where you ran this command. Use this to generate the CSR with the tpm2 openssl provider. The TSS2 PRIVATE KEY file is protected by the TPM and cannot be used on another machine. For more information about TSS2 keys with OpenSSL, see Storing the Private or Public Key.

4. Capture the authentication data for the TSS private key.

```
auth_ecc0=$(echo "$yaml_ecc0" | grep "object-auth" | cut -d' ' -f2-)
```

Step 4: Generate Certificate Signing Request(CSR)

In this step, you'll use the TPM2-protected private key to generate a CSR.

1. Generate a CSR using the TPM2 provider.

```
sudo openssl req -new -provider tpm2 -provider base -key greenkey.pem -passin "pass:
$auth_ecc0" -out "$H"$HOSTNAME".csr
```

When prompted, provide the required information for your CSR, including:

- Country Name (2 letter code)
- State or Province Name
- Locality Name
- Organization Name
- Organizational Unit Name
- Common Name
- Email Address

Alternatively, you can supply an OpenSSL config file for unattended generation. For more information, see OpenSSL Documentation.

2. If you're not generating the CSR on the same machine, copy the generated CSR to a machine that has AWS credentials configured.

Step 5: Create the thing certificate

Create an AWS IoT thing certificate. For more information about how to create a thing certificate, see Create the thing certificate.

Step 6: Import thing certificate to TPM

1. Copy the thing certificate to the device.
2. Add the thing certificate to the Greengrass token.

```
sudo tpm2_ptool addcert --label=greengrass --key-label=greenkey device.pem.crt
```

The available options are the following:

`--help`

Show this help message and exit.

`--label LABEL`

The profile label to remove.

`--key-label KEY_LABEL`

The associated private key label.

`--key-id KEY_ID`

The associated private key id in hex.

`cert`

The x509 PEM certificate to add.

Step 7: Capture the PKCS#11 object URL

We will use the `p11tool` provided from the `gnutls-bin` package to get the PKCS#11 token URL and object URLs.

1. Capture the token URL for the Greengrass token.

```
TOKEN=sudo p11tool --list-token-urls | grep "token=greengrass"
```

2. Get the object URLs for the Greengrass token. Use the same pin you used in Step 3.

```
sudo p11tool --login --list-all "${TOKEN}"
```

Example output:

```
Token 'greengrass' with URL
 'pkcs11:model=SLB9672%00%00%00%00%00%00%00%00;manufacturer=Infineon;serial=0000000000000
 requires user PIN
Enter PIN:
WARNING: Needed CKA_VALUE but didn't find encrypted blob
Object 0:
        URL:
 pkcs11:model=SLB9672%00%00%00%00%00%00%00%00;manufacturer=Infineon;serial=0000000000000
%39%32%37%65%61%64%61%39%31%32%35%31%61%35%37%31;object=greenkey;type=private
        Type: Private key (EC/ECDSA-SECP256R1)
        Label: greenkey
        Flags: CKA_PRIVATE; CKA_NEVER_EXTRACTABLE; CKA_SENSITIVE;
        ID: 39:32:37:65:61:64:61:39:31:32:35:31:61:35:37:31

Object 1:
        URL:
 pkcs11:model=SLB9672%00%00%00%00%00%00%00%00;manufacturer=Infineon;serial=0000000000000
%39%32%37%65%61%64%61%39%31%32%35%31%61%35%37%31;object=greenkey;type=public
        Type: Public key (EC/ECDSA-SECP256R1)
```

```
        Label: greenkey
        ID: 39:32:37:65:61:64:61:39:31:32:35:31:61:35:37:31

Object 2:
        URL:
 pkcs11:model=SLB9672%00%00%00%00%00%00%00%00;manufacturer=Infineon;serial=00000000000000
%39%32%37%65%61%64%61%39%31%32%35%31%61%35%37%31;object=greenkey;type=cert
        Type: X.509 Certificate (EC/ECDSA-SECP256R1)
        Expires: Fri Dec 31 18:59:59 2049
        Label: greenkey
        ID: 39:32:37:65:61:64:61:39:31:32:35:31:61:35:37:31
```

3. Capture the object URL for the private key and the certificate.

Step 8: Configure and install Greengrass with TPM2 support

1. Configure the thing certificate. For more information see Configure the thing certificate.
2. Complete the instructions to Install the AWS IoT Greengrass Core software with the private key and certificate in an HSM in Install the AWS IoT Greengrass Core software. Then, use the following steps to configure your installation to leverage TPM2 through PKCS#11 interface.
3. Verify you have the PKCS#11 provider component downloaded and saved in your Greengrass installer location.
4. Use a text editor to create a configuration file named `config.yaml` to provide to the installer. For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

```
nano GreengrassInstaller/config.yaml
```

5. Copy the following YAML content into the file. This partial configuration file specifies system parameters, Greengrass nucleus parameters, and PKCS#11 provider parameters.

```
---
system:
  certificateFilePath: "pkcs11:model=SW%20%20%20TPM
%00%00%00%00%00%00%00%00;manufacturer=IBM;serial=0000000000000000;token=greengrass;id=
%34%35;object=greenkey;type=cert"
  privateKeyPath: "pkcs11:model=SW%20%20%20TPM
%00%00%00%00%00%00%00%00;manufacturer=IBM;serial=0000000000000000;token=greengrass;id=
%34%35;object=greenkey;type=private"
  rootCaPath: "/greengrass/v2/AmazonRootCA1.pem"
```

```
  rootpath: "/greengrass/v2"
  thingName: "myThing"
services:
  aws.greengrass.Nucleus:
    componentType: "NUCLEUS"
    version: "2.14.0"
    configuration:
      awsRegion: "us-east-1"
      iotRoleAlias: "GreengrassCoreTokenExchangeRoleAlias"
      iotDataEndpoint: "device-data-prefix-ats.iot.us-west-2.amazonaws.com"
      iotCredEndpoint: "device-credentials-prefix.credentials.iot.us-
west-2.amazonaws.com"
  aws.greengrass.crypto.Pkcs11Provider:
    configuration:
      name: "tpm2_pkcs11"
      library: "/usr/lib/x86_64-linux-gnu/pkcs11/libtpm2_pkcs11.so"
      slot: 1
      userPin: "123456"
```

6. Edit the file with the parameters specific for your installation.
   a. Update certificateFilePath and privateKeyPath with the Update certificateFilePath and privateKeyPath with the PKCS#11 URL captured in Step 7.
   b. Update your iotDataEndpoint and iotCredEndpoint based on your AWS IoT endpoint.
   c. In the `aws.greengrass.crypto.Pkcs11Provider` configuration, update the library based on your platform.

> **Note**
>
> The example shown is for X86_64. The file path will be similar for ARM64 device.

7. Complete the Greengrass installation steps in Install the AWS IoT Greengrass Core software.

Step 9: Verify the installation

In this step, you'll verify that Greengrass is running properly with TPM2 integration.

1. Check the Greengrass service status.

```
sudo systemctl status greengrass.service
```

2. View the Greengrass logs to ensure there are no errors.

```
sudo tail -f /greengrass/v2/logs/greengrass.log
```

3. Verify that your device appears as connected in the AWS IoT console.
   a. Sign in to the AWS IoT Greengrass console.
   b. Under **Manage**, expand **Greengrass devices**, and choose **Core devices**.
   c. Confirm your device is connected. The device status will show HEALTHY if it is connected. For more information, see Check Greengrass core device status.

Troubleshooting

If you encounter issues during the setup or operation of your TPM2-enabled Greengrass device, try the following troubleshooting steps.

- Check the main Greengrass log file.

```
sudo tail -f /greengrass/v2/logs/greengrass.log
```

- Verify PKCS#11 provider configuration.

```
sudo cat /greengrass/v2/config/effectiveConfig.yaml
```

- Ensure the TPM2 service is running.

```
sudo systemctl status tpm2-abrmd.service
```

- Verify that the TPM2 key is accessible.

```
sudo pkcs11-tool –module /usr/lib/x86_64-linux-gnu/pkcs11/libtpm2_pkcs11.so -l -p
 123456 –list-objects
```

- If your OS is set up with full disk encryption with TPM2 storage root keys such as Clevis or systemd-cryptenroll, verify you aren't using the same persistent handle used by these tools. Using the same persistent handle might affect your disk encryption mechanism. To check all created and used persistent handles, run the following command

```
sudo tpm2_getcap handles-persistent
```

Next steps

Now that you've successfully integrated your Greengrass core device with TPM2, you can:

- Deploy components to your secure Greengrass device. For more information, see Deploy AWS IoT Greengrass components to devices
- Set up additional Greengrass devices with TPM2 integration.

For more information about security on Greengrass devices, see Security in AWS IoT Greengrass.

Tutorial: Secure AWS IoT Greengrass Nucleus Lite with Trusted Platform Module (TPM)

This tutorial explains how to enable and configure Trusted Platform Module (TPM) support for AWS IoT Greengrass nucleus lite. TPM provides hardware-based root of trust for secure key storage. This security feature protects cryptographic operations and sensitive credentials, enhancing device security and integrity.

When you complete this integration, your AWS IoT Greengrass core device will use TPM-protected private keys for its identity and communication with AWS IoT services.

For more information about security on AWS IoT Greengrass devices, see Security in AWS IoT Greengrass.

Prerequisites

To complete this tutorial, you need the following:

- A Linux-compatible device with TPM 2.0 hardware or NitroTPM
- A developer machine with Greengrass Nucleus Lite installed. For more information, see Install the AWS IoT Greengrass Core software (console).
- Instructions in this tutorial are defined for Ubuntu 24.04 LTS.
- Any Linux distribution that can support the Linux TPM2 Software Stack can support this mechanism.
- A developer machine with AWS CLI installed and configured with permissions to:
  - Create and manage AWS IoT resources

  - Create and manage IAM roles and policies
- Root or sudo privileges on the device.

This tutorial contains instructions on how to use the TPM2 chip as a Hardware Security Module (HSM) to create a private key and CSR which is used to create the AWS IoT thing certificate.

Step 1: Set up a NitroTPM instance

1. Set up a NitroTPM instance. For more information, see NitroTPM.
2. Launch final instance using the custom AMI created in the previous step.

   > **Important**
   >
   > When connecting with SSH, use the `ubuntu` user instead of `root`.

3. Check that the TPM device is present and working by running the following command:

   ```
   ls -la /dev/tpm*
   ```

   You should see `/dev/tpm0` and `/dev/tpmrm0` devices.

Step 2: Install and configure TPM tools

1. Install the required packages by running the following command:

   ```
   sudo apt update
   sudo apt install tpm2-openssl tpm2-tools tpm2-abrmd libtss2-tcti-tabrmd0
   ```

2. Verify the TPM device's permissions by running the following command:

   ```
   ls -l /dev/tpm0    # Should be owned by tss:root with permissions 0660
   ls -l /dev/tpmrm0  # Should be owned by tss:tss with permissions 0660
   ```

Step 3: Configure the OpenSSL TPM2 provider

1. Edit the OpenSSL configuration file:

```
sudo vi /etc/ssl/openssl.cnf
```

2. Add the following configuration:

```
[openssl_init]
providers = provider_sect

[provider_sect]
default = default_sect
tpm2 = tpm2_sect

[default_sect]
activate = 1

[tpm2_sect]
identity = tpm2
module = /usr/local/lib64/tpm2.so
activate = 1
```

3. Adjust the module path as necessary. You can find the correct path using:

```
find /usr -name "tpm2.so"
```

Step 4: Generate persistent TPM keys

1. Create a primary key by running the following command:

```
sudo tpm2_createprimary -C o -c primary.ctx
```

2. Create an ECC key object by running the following command:

```
sudo tpm2_create -C primary.ctx -g sha256 -G ecc256 -r device.priv -u device.pub
```

3. Load the key by running the following command:

```
sudo tpm2_load -C primary.ctx -r device.priv -u device.pub -c device.ctx
```

4. Make the key persistent by running the following command:

```
sudo tpm2_evictcontrol -C o -c device.ctx 0x81000002
```

This creates a persistent key with the handle (like `0x81000002`).

Step 5: Generate Certificate Signing Request (CSR)

In this step, you'll use the TPM2-protected private key to generate a Certificate Signing Request (CSR).

1. Generate a CSR using the TPM key:

```
openssl req -new -provider tpm2 -key "handle:0x81000002" \
    -out device.csr \
    -subj "/CN=TPMThing"
```

2. Replace `0x81000002` with your chosen handle value and `TPMThing` with your desired thing name.

Step 6: Create the certificate from CSR

1. On your development computer, create a folder where you downloaded the certificate for the AWS IoT thing.

```
mkdir greengrass-v2-certs
```

2. Use the CSR file to create and download the certificate for the AWS IoT thing to your development computer.

```
aws iot create-certificate-from-csr \
  --set-as-active \
  --certificate-signing-request file://path_to_device.csr \
  --certificate-pem-outfile greengrass-v2-certs/device.pem.crt
```

If the request succeeds, the response looks similar to the following example:

```
{
"certificateArn": "arn:aws:iot:us-west-2:123456789012:cert/
aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4",
```

```
  "certificateId":
 "aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4",
  "certificatePem": "-----BEGIN CERTIFICATE-----
MIICiTCCAfICCQD6m7oRw0uXOjANBgkqhkiG9w
 0BAQUFADCBiDELMAkGA1UEBhMCVVMxCzAJBgNVBAgTAldBMRAwDgYDVQQHEwdTZ
 WF0dGxlMQ8wDQYDVQQKEwZBbWF6b24xFDASBgNVBAsTC0lBTSBDb25zb2xlMRIw
 EAYDVQQDEwlUZXN0Q2lsYWMxHzAdBgkqhkiG9w0BCQEWEG5vb25lQGFtYXpvbi5
 jb20wHhcNMTEwNDI1MjA0NTIxWhcNMTIwNDI0MjA0NTIxWjCBiDELMAkGA1UEBh
 MCVVMxCzAJBgNVBAgTAldBMRAwDgYDVQQHEwdTZWF0dGxlMQ8wDQYDVQQKEwZBb
 WF6b24xFDASBgNVBAsTC0lBTSBDb25zb2xlMRIwEAYDVQQDEwlUZXN0Q2lsYWMx
 HzAdBgkqhkiG9w0BCQEWEG5vb25lQGFtYXpvbi5jb20wgZ8wDQYJKoZIhvcNAQE
 BBQADgY0AMIGJAoGBAMaK0dn+a4GmWIWJ21uUSfwfEvySWtC2XADZ4nB+BLYgVI
 k60CpiwsZ3G93vUEIO3IyNoH/f0wYK8m9TrDHudUZg3qX4waLG5M43q7Wgc/MbQ
 ITxOUSQv7c7ugFFDzQGBzZswY6786m86gpEIbb3OhjZnzcvQAaRHhdlQWIMm2nr
 AgMBAAEwDQYJKoZIhvcNAQEFBQADgYEAtCu4nUhVVxYUntneD9+h8Mg9q6q+auN
 KyExzyLwaxlAoo7TJHidbtS4J5iNmZgXL0FkbFFBjvSfpJIlJ00zbhNYS5f6Guo
 EDmFJl0ZxBHjJnyp378OD8uTs7fLvjx79LjSTbNYiytVbZPQUQ5Yaxu2jXnimvw
 3rrszlaEXAMPLE=
-----END CERTIFICATE-----"
}
```

Step 7: Set up Greengrass Nucleus Lite with TPM support

To enable TPM support in Greengrass Nucleus Lite, make the following changes:

1. Configure user permissions by adding the `ggcore` user to the `tss` group for TPM access:

   ```
   sudo usermod -a -G tss ggcore
   ```

2. Update the credentials directory by completing the following steps:

   - Remove the private key file from the credentials directory.
     - Since we are using persistent TPM keys, no private key file needs to be copied.
   - Copy the new `device.pem.crt` into that credentials directory.

3. Edit `config.yaml` with the following TPM-specific configuration:

   ```
   system:
       privateKeyPath: "handle:0x81000002" # Use your chosen handle
       certificateFilePath: "" # Replace with the path of device.pem.crt
       ...
   ```

4. Reboot your Greengrass Nucleus Lite by running the following command:

```
systemctl restart greengrass-lite.target
```

Troubleshooting

If you encounter issues during the setup or operation of your TPM2-enabled AWS IoT Greengrass device, try the following troubleshooting steps:

TPM device is not found (when using NitroTPM)

If `/dev/tpm0` is not present, do the following steps:

1. Verify you're using a NitroTPM-supported instance type.
2. Ensure the AMI was created with `--tpm-support v2.0`.
3. Verify that the instance was launched from the custom AMI.

Permission denied errors

If you encounter TPM access errors, do the following:

1. Verify that the user is in the `tss` group: `groups $USER`.
2. Check the TPM device's permissions using the following command:

```
ls -l /dev/tpm*
```

3. Verify you logged out and logged back in after adding to the `tss` group.

OpenSSL provider issues

If TPM provider is not found, do the following:

1. Verify `tpm2.so` path in the `/etc/ssl/openssl.cnf`.
2. Check provider installation by running the following command:

```
openssl list -providers
```

3. Verify the `tpm2-openssl` package is properly installed.

Next steps

Now that you've successfully integrated your AWS IoT Greengrass core device with TPM2, you can:

- Deploy components to your secure AWS IoT Greengrass device
- Set up additional AWS IoT Greengrass devices with TPM2 integration

For more information about security on AWS IoT Greengrass devices, see Security in AWS IoT Greengrass.

Tutorial: Get started with SageMaker AI Edge Manager

> **Important**
>
> SageMaker AI Edge Manager was discontinued on April 26th, 2024. For more information about continuing to deploy your models to edge devices, see SageMaker AI Edge Manager end of life.

Amazon SageMaker AI Edge Manager is a software agent that runs on edge devices. SageMaker AI Edge Manager provides model management for edge devices so that you can package and use Amazon SageMaker AI Neo-compiled models directly on Greengrass core devices. By using SageMaker AI Edge Manager, you can also sample model input and output data from your core devices, and send that data to the AWS Cloud for monitoring and analysis. For more information about how SageMaker AI Edge Manager works on Greengrass core devices, see Use Amazon SageMaker AI Edge Manager on Greengrass core devices.

This tutorial shows you how to get started using SageMaker AI Edge Manager with AWS-provided sample components on an existing core device. These sample components use the SageMaker AI Edge Manager component as a dependency to deploy the Edge Manager agent, and perform inference using pre-trained models that were compiled using SageMaker AI Neo. For more information about the SageMaker AI Edge Manager agent, see SageMaker AI Edge Manager in the *Amazon SageMaker AI Developer Guide*.

To set up and use the SageMaker AI Edge Manager agent on an existing Greengrass core device, AWS provides example code that you can use to create the following sample inference and model components.

- **Image classification**
  - `com.greengrass.SageMakerEdgeManager.ImageClassification`
  - `com.greengrass.SageMakerEdgeManager.ImageClassification.Model`
- **Object detection**
  - `com.greengrass.SageMakerEdgeManager.ObjectDetection`
  - `com.greengrass.SageMakerEdgeManager.ObjectDetection.Model`

This tutorial shows you how to deploy the sample components and the SageMaker AI Edge Manager agent.

Topics

- Prerequisites
- Set up your Greengrass core device in SageMaker AI Edge Manager
- Create the sample components
- Run sample image classification inference

Prerequisites

To complete this tutorial, you must meet the following prerequisites:

- A Greengrass core device running on Amazon Linux 2, a Debian-based Linux platform (x86_64 or Armv8), or Windows (x86_64). If you don't have one, see Tutorial: Getting started with AWS IoT Greengrass V2.
- Python 3.6 or later, including `pip` for your version of Python, installed on your core device.
- The OpenGL API GLX runtime (`libgl1-mesa-glx`) installed on your core device.
- An AWS Identity and Access Management (IAM) user with administrator permissions.
- An internet-enabled Windows, Mac, or Unix-like development computer that meets the following requirements:
  - Python 3.6 or later installed.
  - AWS CLI installed and configured with your IAM administrator user credentials. For more information, see Installing the AWS CLI and Configuring the AWS CLI.
- The following S3 buckets created in the same AWS account and AWS Region as your Greengrass core device:

- An S3 bucket to store the artifacts that are included in the sample inference and model components. This tutorial uses amzn-s3-demo-bucket1 to refer to this bucket.
- An S3 bucket that you associate with your SageMaker AI edge device fleet. SageMaker AI Edge Manager requires an S3 bucket to create the edge device fleet, and to store sample data from running inference on your device. This tutorial uses amzn-s3-demo-bucket2 to refer to this bucket.

  For information about creating S3 buckets, see Getting started with Amazon S3.

- The Greengrass device role configured with the following:
  - A trust relationship that allows `credentials.iot.amazonaws.com` and `sagemaker.amazonaws.com` to assume the role, as shown in the following IAM policy example.

    ```
    {
      "Version": "2012-10-17",
      "Statement": [
        {
          "Effect": "Allow",
          "Principal": {
            "Service": "credentials.iot.amazonaws.com"
           },
          "Action": "sts:AssumeRole"
        },
        {
          "Effect": "Allow",
          "Principal": {
            "Service": "sagemaker.amazonaws.com"
          },
          "Action": "sts:AssumeRole"
        }
      ]
    }
    ```

  - The AmazonSageMakerEdgeDeviceFleetPolicy IAM managed policy.
  - The AmazonSageMakerFullAccess IAM managed policy.
  - The `s3:GetObject` action for the S3 bucket that contains your component artifacts, as shown in the following IAM policy example.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
  {
    "Action": [
      "s3:GetObject"
    ],
    "Resource": [
      "arn:aws:s3:::amzn-s3-demo-bucket1/*"
    ],
    "Effect": "Allow"
  }
  ]
}
```

Set up your Greengrass core device in SageMaker AI Edge Manager

Edge device fleets in SageMaker AI Edge Manager are collections of logically grouped devices. To use SageMaker AI Edge Manager with AWS IoT Greengrass, you must create an edge device fleet that uses the same AWS IoT role alias as the Greengrass core device to which you deploy the SageMaker AI Edge Manager agent. Then, you must register the core device as part of that fleet.

Topics

- Create an edge device fleet
- Register your Greengrass core device

Create an edge device fleet

To create an edge device fleet (console)

1. In the Amazon SageMaker AI console, choose **Edge Manager**, and then choose **Edge device fleets**.
2. On the **Device fleets** page, choose **Create device fleet**.
3. Under **Device fleet properties**, do the following:

   - For **Device fleet name**, enter a name for your device fleet.

- For **IAM role**, enter the Amazon Resource Name (ARN) of the AWS IoT role alias that you specified when setting up your Greengrass core device.
- Disable the **Create IAM role alias** toggle.

4. Choose **Next**.
5. Under **Output configuration**, for **S3 bucket URI**, enter the URI of the S3 bucket that you want to associate with the device fleet.
6. Choose **Submit**.

Register your Greengrass core device

To register your Greengrass core device as an edge device (console)

1. In the Amazon SageMaker AI console, choose **Edge Manager**, and then choose **Edge devices**.
2. On the **Devices** page, choose **Register devices**.
3. Under **Device properties**, for **Device fleet name**, enter the name of the device fleet that you created, and then choose **Next**.
4. Choose **Next**.
5. Under **Device source**, for **Device name**, enter the AWS IoT thing name of your Greengrass core device.
6. Choose **Submit**.

Create the sample components

To help you get started using the SageMaker AI Edge Manager component, AWS provides a Python script on GitHub that creates the sample inference and model components and uploads them to the AWS Cloud for you. Complete the following steps on a development computer.

To create the sample components

1. Download the AWS IoT Greengrass component examples repository on GitHub to your development computer.
2. Navigate to the downloaded `/machine-learning/sagemaker-edge-manager` folder.

```
cd download-directory/machine-learning/sagemaker-edge-manager
```

3. Run the following command to create and upload the sample components to the AWS Cloud.

```
python3 create_components.py -r region -b amzn-s3-demo-bucket
```

Replace *region* with the AWS Region where you created your Greengrass core device, and replace amzn-s3-demo-bucket1 with the name of the S3 bucket to store your component artifacts.

> **Note**
>
> By default, the script creates sample components for both image classification and object detection inference. To create components for only a specific type of inference, specify the `-i ImageClassification | ObjectDetection` argument.

Sample inference and model components for use with SageMaker AI Edge Manager are now created in your AWS account. To see the sample components in the AWS IoT Greengrass console, choose **Components**, and then under **My components**, search for the following components:

- `com.greengrass.SageMakerEdgeManager.ImageClassification`
- `com.greengrass.SageMakerEdgeManager.ImageClassification.Model`
- `com.greengrass.SageMakerEdgeManager.ObjectDetection`
- `com.greengrass.SageMakerEdgeManager.ObjectDetection.Model`

Run sample image classification inference

To run image classification inference using the AWS-provided sample components and the SageMaker AI Edge Manager agent, you must deploy these components to your core device. Deploying these components downloads a SageMaker AI Neo-compiled pre-trained Resnet-50 model and installs the SageMaker AI Edge Manager agent on your device. The SageMaker AI Edge Manager agent loads the model and publishes inference results on the gg/`sageMakerEdgeManager/image-classification` topic. To view these inference results, use the AWS IoT MQTT client in the AWS IoT console to subscribe to this topic.

Topics

- Subscribe to the notifications topic

- Deploy the sample components
- View inference results

Subscribe to the notifications topic

In this step, you configure the AWS IoT MQTT client in the AWS IoT console to watch MQTT messages published by the sample inference component. By default, the component publishes inference results on the `gg/sageMakerEdgeManager/image-classification` topic. Subscribe to this topic before you deploy the component to your Greengrass core device to see the inference results when the component runs for the first time.

To subscribe to the default notifications topic

1. In the AWS IoT console navigation menu, choose **Test, MQTT test client**.
2. Under **Subscribe to a topic**, in the **Topic name** box, enter **`gg/sageMakerEdgeManager/image-classification`**.
3. Choose **Subscribe**.

Deploy the sample components

In this step, you configure and deploy the following components to your core device:

- `aws.greengrass.SageMakerEdgeManager`
- `com.greengrass.SageMakerEdgeManager.ImageClassification`
- `com.greengrass.SageMakerEdgeManager.ImageClassification.Model`

To deploy your components (console)

1. In the AWS IoT Greengrass console navigation menu, choose **Deployments**, and then choose the deployment for your target device that you want to revise.
2. On the deployment page, choose **Revise**, and then choose **Revise deployment**.
3. On the **Specify target** page, choose **Next**.
4. On the **Select components** page, do the following:

    a. Under **My components**, select the following components:

- `com.greengrass.SageMakerEdgeManager.ImageClassification`
- `com.greengrass.SageMakerEdgeManager.ImageClassification.Model`

b. Under **Public components**, turn off the **Show only selected components** toggle, and then select the `aws.greengrass.SageMakerEdgeManager` component.

c. Choose **Next**.

5. On the **Configure components** page, select the `aws.greengrass.SageMakerEdgeManager` component and do the following.

a. Choose **Configure component**.

b. Under **Configuration update**, in **Configuration to merge**, enter the following configuration.

```
{
    "DeviceFleetName": "device-fleet-name",
    "BucketName": "amzn-s3-demo-bucket"
}
```

Replace *`device-fleet-name`* with the name of the edge device fleet that you created, and replace *`amzn-s3-demo-bucket`* with the name of the S3 bucket that is associated with your device fleet.

c. Choose **Confirm**, and then choose **Next**.

6. On the **Configure advanced settings** page, keep the default configuration settings, and choose **Next**.

7. On the **Review** page, choose **Deploy**

To deploy your components (AWS CLI)

1. On your development computer, create a `deployment.json` file to define the deployment configuration for your SageMaker AI Edge Manager components. This file should look like the following example.

```
{
  "targetArn":"targetArn",
  "components": {
    "aws.greengrass.SageMakerEdgeManager": {
      "componentVersion": "1.0.x",
```

```
      "configurationUpdate": {
        "merge": "{\"DeviceFleetName\":\"device-fleet-name\",\"BucketName\":\"amzn-
s3-demo-bucket2\"}"
      }
    },
    "com.greengrass.SageMakerEdgeManager.ImageClassification": {
      "componentVersion": "1.0.x",
      "configurationUpdate": {
      }
    },
    "com.greengrass.SageMakerEdgeManager.ImageClassification.Model": {
      "componentVersion": "1.0.x",
      "configurationUpdate": {
      }
    },
  }
}
```

- In the `targetArn` field, replace *`targetArn`* with the Amazon Resource Name (ARN) of the thing or thing group to target for the deployment, in the following format:
  - Thing: `arn:aws:iot:`*`region`*`:`*`account-id`*`:thing/`*`thingName`*
  - Thing group: `arn:aws:iot:`*`region`*`:`*`account-id`*`:thinggroup/`*`thingGroupName`*
- In the `merge` field, replace *`device-fleet-name`* with the name of the edge device fleet that you created. Then, replace *`amzn-s3-demo-bucket2`* with the name of the S3 bucket that is associated with your device fleet.
- Replace the component versions for each component with the latest available version.

2. Run the following command to deploy the components on the device:

```
aws greengrassv2 create-deployment \
    --cli-input-json file://path/to/deployment.json
```

The deployment can take several minutes to complete. In the next step, check the component log to verify that the deployment completed successfully and to view the inference results.

View inference results

After you deploy the components, you can view the inference results in the component log on your Greengrass core device and in the AWS IoT MQTT client in the AWS IoT console. To subscribe to the topic on which the component publishes inference results, see Subscribe to the notifications topic.

- **AWS IoT MQTT client**—To view the results that the inference component publishes on the default notifications topic, complete the following steps:

  1. In the AWS IoT console navigation menu, choose **Test, MQTT test client**.
  2. Under **Subscriptions**, choose **`gg/sageMakerEdgeManager/image-classification`**.

- **Component log**—To view the inference results in the component log, run the following command on your Greengrass core device.

```
sudo tail -f /greengrass/v2/logs/
com.greengrass.SageMakerEdgeManager.ImageClassification.log
```

If you can't see inference results in the component log or in the MQTT client, the deployment failed or didn't reach the core device. This can occur if your core device isn't connected to the internet or doesn't have the right permissions to run the component. Run the following command on your core device to view the AWS IoT Greengrass Core software log file. This file includes logs from the Greengrass core device's deployment service.

```
sudo tail -f /greengrass/v2/logs/greengrass.log
```

For more information, see Troubleshooting machine learning inference.

Tutorial: Perform sample image classification inference using TensorFlow Lite

This tutorial shows you how to use the TensorFlow Lite image classification inference component to perform sample image classification inference on a Greengrass core device. This component includes the following component dependencies:

- TensorFlow Lite image classification model store component
- TensorFlow Lite runtime component

When you deploy this component, it downloads a pre-trained MobileNet v1 model and installs the TensorFlow Lite runtime and its dependencies. This component publishes inference results on the `ml/tflite/image-classification` topic. To view these inference results, use the AWS IoT MQTT client in the AWS IoT console to subscribe to this topic.

In this tutorial you deploy the sample inference component to perform image classification on the sample image that is provided by AWS IoT Greengrass. After you complete this tutorial, you can complete Tutorial: Perform sample image classification inference on images from a camera using TensorFlow Lite, which shows you how to modify the sample inference component to perform image classification on images from a camera locally on a Greengrass core device.

For more information about machine learning on Greengrass devices, see Perform machine learning inference.

Topics

- Prerequisites
- Step 1: Subscribe to the default notifications topic
- Step 2: Deploy the TensorFlow Lite image classification component
- Step 3: View inference results
- Next steps

Prerequisites

To complete this tutorial, you need the following:

- A Linux Greengrass core device. If you don't have one, see Tutorial: Getting started with AWS IoT Greengrass V2. The core device must meet the following requirements:
  - On Greengrass core devices running Amazon Linux 2 or Ubuntu 18.04, GNU C Library (glibc) version 2.27 or later installed on the device.
  - On Armv7l devices, such as Raspberry Pi, dependencies for OpenCV-Python installed on the device. Run the following command to install the dependencies.

```
sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev
 libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
```

- Raspberry Pi devices that run Raspberry Pi OS Bullseye must meet the following requirements:
  - NumPy 1.22.4 or later installed on the device. Raspberry Pi OS Bullseye includes an earlier version of NumPy, so you can run the following command to upgrade NumPy on the device.

    ```
    pip3 install --upgrade numpy
    ```
  - The legacy camera stack enabled on the device. Raspberry Pi OS Bullseye includes a new camera stack that is enabled by default and isn't compatible, so you must enable the legacy camera stack.

    To enable the legacy camera stack

    1. Run the following command to open the Raspberry Pi configuration tool.

       ```
       sudo raspi-config
       ```
    2. Select **Interface Options**.
    3. Select **Legacy camera** to enable the legacy camera stack.
    4. Reboot the Raspberry Pi.

Step 1: Subscribe to the default notifications topic

In this step, you configure the AWS IoT MQTT client in the AWS IoT console to watch MQTT messages published by the TensorFlow Lite image classification component. By default, the component publishes inference results on the `ml/tflite/image-classification` topic. Subscribe to this topic before you deploy the component to your Greengrass core device to see the inference results when the component runs for the first time.

To subscribe to the default notifications topic

1. In the AWS IoT console navigation menu, choose **Test, MQTT test client**.
2. Under **Subscribe to a topic**, in the **Topic name** box, enter **`ml/tflite/image-classification`**.
3. Choose **Subscribe**.

Step 2: Deploy the TensorFlow Lite image classification component

In this step, you deploy the TensorFlow Lite image classification component to your core device:

To deploy the TensorFlow Lite image classification component (console)

1. In the AWS IoT Greengrass console navigation menu, choose **Components**.
2. On the **Components** page, on the **Public components** tab, choose `aws.greengrass.TensorFlowLiteImageClassification`.
3. On the **aws.greengrass.TensorFlowLiteImageClassification** page, choose **Deploy**.
4. From **Add to deployment**, choose one of the following:
   a. To merge this component to an existing deployment on your target device, choose **Add to existing deployment**, and then select the deployment that you want to revise.
   b. To create a new deployment on your target device, choose **Create new deployment**. If you have an existing deployment on your device, choosing this step replaces the existing deployment.
5. On the **Specify target** page, do the following:
   a. Under **Deployment** information, enter or modify the friendly name for your deployment.
   b. Under **Deployment targets**, select a target for your deployment, and choose **Next**. You cannot change the deployment target if you are revising an existing deployment.
6. On the **Select components** page, under **Public components**, verify that the `aws.greengrass.TensorFlowLiteImageClassification` component is selected, and choose **Next**.
7. On the **Configure components** page, keep the default configuration settings, and choose **Next**.
8. On the **Configure advanced settings** page, keep the default configuration settings, and choose **Next**.
9. On the **Review** page, choose **Deploy**

To deploy the TensorFlow Lite image classification component (AWS CLI)

1. Create a `deployment.json` file to define the deployment configuration for the TensorFlow Lite image classification component. This file should look like the following:

```
{
  "targetArn":"targetArn",
  "components": {
    "aws.greengrass.TensorFlowLiteImageClassification": {
      "componentVersion": 2.1.0,
      "configurationUpdate": {
      }
    }
  }
}
```

- In the `targetArn` field, replace *`targetArn`* with the Amazon Resource Name (ARN) of the thing or thing group to target for the deployment, in the following format:
  - Thing: `arn:aws:iot:`*`region`*`:`*`account-id`*`:thing/`*`thingName`*
  - Thing group: `arn:aws:iot:`*`region`*`:`*`account-id`*`:thinggroup/`*`thingGroupName`*
- This tutorial uses component version 2.1.0. In the `aws.greengrass.TensorFlowLiteObjectDetection` component object, replace *`2.1.0`* to use a different version of the TensorFlow Lite object detection component.

2. Run the following command to deploy the TensorFlow Lite image classification component on the device:

```
aws greengrassv2 create-deployment \
    --cli-input-json file://path/to/deployment.json
```

The deployment can take several minutes to complete. In the next step, check the component log to verify that the deployment completed successfully and to view the inference results.

Step 3: View inference results

After you deploy the component, you can view the inference results in the component log on your Greengrass core device and in the AWS IoT MQTT client in the AWS IoT console. To subscribe to the topic on which the component publishes inference results, see Step 1: Subscribe to the default notifications topic.

- **AWS IoT MQTT client**—To view the results that the inference component publishes on the default notifications topic, complete the following steps:

1. In the AWS IoT console navigation menu, choose **Test, MQTT test client**.
2. Under **Subscriptions**, choose **`ml/tflite/image-classification`**.

   You should see messages similar to the following example.

```
{
  "timestamp": "2021-01-01 00:00:00.000000",
  "inference-type": "image-classification",
  "inference-description": "Top 5 predictions with score 0.3 or above ",
  "inference-results": [
    {
      "Label": "cougar, puma, catamount, mountain lion, painter, panther, Felis
 concolor",
      "Score": "0.5882352941176471"
    },
    {
      "Label": "Persian cat",
      "Score": "0.5882352941176471"
    },
    {
      "Label": "tiger cat",
      "Score": "0.5882352941176471"
    },
    {
      "Label": "dalmatian, coach dog, carriage dog",
      "Score": "0.5607843137254902"
    },
    {
      "Label": "malamute, malemute, Alaskan malamute",
      "Score": "0.5450980392156862"
    }
  ]
}
```

- **Component log**—To view the inference results in the component log, run the following command on your Greengrass core device.

```
sudo tail -f /greengrass/v2/logs/aws.greengrass.TensorFlowLiteImageClassification.log
```

  You should see results similar to the following example.

```
2021-01-01 00:00:00.000000 [INFO] (Copier)
 aws.greengrass.TensorFlowLiteImageClassification: stdout. Publishing results to the
 IoT core....
 {scriptName=services.aws.greengrass.TensorFlowLiteImageClassification.lifecycle.Run.script,
 serviceName=aws.greengrass.TensorFlowLiteImageClassification, currentState=RUNNING}

2021-01-01 00:00:00.000000 [INFO] (Copier)
 aws.greengrass.TensorFlowLiteImageClassification: stdout. {"timestamp":
 "2021-01-01 00:00:00.000000", "inference-type": "image-classification", "inference-
description": "Top 5 predictions with score 0.3 or above ", "inference-results":
 [{"Label": "cougar, puma, catamount, mountain lion, painter, panther, Felis
 concolor", "Score": "0.5882352941176471"}, {"Label": "Persian cat", "Score":
 "0.5882352941176471"}, {"Label": "tiger cat", "Score": "0.5882352941176471"},
 {"Label": "dalmatian, coach dog, carriage dog", "Score": "0.5607843137254902"},
 {"Label": "malamute, malemute, Alaskan malamute", "Score": "0.5450980392156862"}]}.
 {scriptName=services.aws.greengrass.TensorFlowLiteImageClassification.lifecycle.Run.script,
 serviceName=aws.greengrass.TensorFlowLiteImageClassification, currentState=RUNNING}
```

If you can't see inference results in the component log or in the MQTT client, the deployment failed or didn't reach the core device. This can occur if your core device isn't connected to the internet or doesn't have the right permissions to run the component. Run the following command on your core device to view the AWS IoT Greengrass Core software log file. This file includes logs from the Greengrass core device's deployment service.

```
sudo tail -f /greengrass/v2/logs/greengrass.log
```

For more information, see Troubleshooting machine learning inference.

Next steps

If you have a Greengrass core device with a supported camera interface, you can complete Tutorial: Perform sample image classification inference on images from a camera using TensorFlow Lite, which shows you how to modify the sample inference component to perform image classification on images from a camera.

To further explore the configuration of the sample TensorFlow Lite image classification inference component, try the following:

- Modify the `InferenceInterval` configuration parameter to change how often the inference code runs.
- Modify the `ImageName` and `ImageDirectory` configuration parameters in the inference component configuration to specify a custom image to use for inference.

For information about customizing the configuration of public components or creating custom machine learning components, see Customize your machine learning components.

Tutorial: Perform sample image classification inference on images from a camera using TensorFlow Lite

This tutorial shows you how to use the TensorFlow Lite image classification inference component to perform sample image classification inference on images from a camera locally on a Greengrass core device. This component includes the following component dependencies:

- TensorFlow Lite image classification model store component
- TensorFlow Lite runtime component

> **Note**
>
> This tutorial accesses the camera module for Raspberry Pi or NVIDIA Jetson Nano devices, but AWS IoT Greengrass supports other devices on Armv7l, Armv8, or x86_64 platforms. To set up a camera for a different device, consult the relevant documentation for your device.

For more information about machine learning on Greengrass devices, see Perform machine learning inference.

Topics

- Prerequisites
- Step 1: Configure the camera module on your device
- Step 2: Verify your subscription to the default notifications topic
- Step 3: Modify the TensorFlow Lite image classification component configuration and deploy it
- Step 4: View inference results
- Next steps

Prerequisites

To complete this tutorial, you must first complete Tutorial: Perform sample image classification inference using TensorFlow Lite.

You also need the following:

- A Linux Greengrass core device with a camera interface. This tutorial accesses the camera module on one the following supported devices:
  - Raspberry Pi running Raspberry Pi OS (previously called Raspbian)
  - NVIDIA Jetson Nano

  For information about setting up a Greengrass core device, see Tutorial: Getting started with AWS IoT Greengrass V2.

  The core device must meet the following requirements:
  - On Greengrass core devices running Amazon Linux 2 or Ubuntu 18.04, GNU C Library (glibc) version 2.27 or later installed on the device.
  - On Armv7l devices, such as Raspberry Pi, dependencies for OpenCV-Python installed on the device. Run the following command to install the dependencies.

    ```
    sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev
     libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
    ```
  - Raspberry Pi devices that run Raspberry Pi OS Bullseye must meet the following requirements:
    - NumPy 1.22.4 or later installed on the device. Raspberry Pi OS Bullseye includes an earlier version of NumPy, so you can run the following command to upgrade NumPy on the device.

      ```
      pip3 install --upgrade numpy
      ```
    - The legacy camera stack enabled on the device. Raspberry Pi OS Bullseye includes a new camera stack that is enabled by default and isn't compatible, so you must enable the legacy camera stack.

      To enable the legacy camera stack

      1. Run the following command to open the Raspberry Pi configuration tool.

         ```
         sudo raspi-config
         ```

2. Select **Interface Options**.
3. Select **Legacy camera** to enable the legacy camera stack.
4. Reboot the Raspberry Pi.

- For Raspberry Pi or NVIDIA Jetson Nano devices, Raspberry Pi Camera Module V2 - 8 megapixel, 1080p. To learn how to set up the camera, see Connecting the camera in the Raspberry Pi documentation.

Step 1: Configure the camera module on your device

In this step, you install and enable the camera module for your device. Run the following commands on the device.

Raspberry Pi (Armv7l)

1. Install the `picamera` interface for the camera module. Run the following command to install the camera module and the other Python libraries that are required for this tutorial.

   ```
   sudo apt-get install -y python3-picamera
   ```

2. Verify that Picamera installed successfully.

   ```
   sudo -u ggc_user bash -c 'python3 -c "import picamera"'
   ```

   If the output doesn't contain errors, the validation is successful.

   > **Note**
   >
   > If the Python executable file that is installed on your device is `python3.7`, use `python3.7` instead of `python3` for the commands in this tutorial. Make sure that your pip installation maps to the correct `python3.7` or `python3` version to avoid dependency errors.

3. Reboot the device.

   ```
   sudo reboot
   ```

4. Open the Raspberry Pi configuration tool.

```
sudo raspi-config
```

5. Use the arrow keys to open **Interfacing Options** and enable the camera interface. If prompted, allow the device to reboot.
6. Run the following command to test the camera setup.

   ```
   raspistill -v -o test.jpg
   ```

   This opens a preview window on the Raspberry Pi, saves a picture named `test.jpg` to your current directory, and displays information about the camera in the Raspberry Pi terminal.
7. Run the following command to create a symlink to enable the inference component to access your camera from the virtual environment that is created by the runtime component.

   ```
   sudo ln -s /usr/lib/python3/dist-packages/picamera "MLRootPath/
   greengrass_ml_tflite_venv/lib/python3.7/site-packages"
   ```

   The default value for *MLRootPath* for this tutorial is */greengrass/v2*`/work/variant.TensorFlowLite/greengrass_ml`. The `greengrass_ml_tflite_venv` folder in this location is created when you deploy the inference component for the first time in Tutorial: Perform sample image classification inference using TensorFlow Lite.

Jetson Nano (Armv8)

1. Run the following command to test the camera setup.

   ```
   gst-launch-1.0 nvarguscamerasrc num-buffers=1 ! "video/x-raw(memory:NVMM),
    width=1920, height=1080, format=NV12, framerate=30/1" ! nvjpegenc ! filesink
    location=test.jpg
   ```

   This captures and saves an image named `test.jpg` to your current directory.
2. (Optional) Reboot the device. If you encounter issues when you run the `gst-launch` command in the previous step, rebooting your device might resolve those issues.

   ```
   sudo reboot
   ```

> **Note**
>
> For Armv8 (AArch64) devices, such as a Jetson Nano, you don't need to create a symlink to enable the inference component to access the camera from the virtual environment that is created by the runtime component.

Step 2: Verify your subscription to the default notifications topic

In Tutorial: Perform sample image classification inference using TensorFlow Lite, you configured the AWS IoT MQTT client is configured in the AWS IoT console to watch MQTT messages published by the TensorFlow Lite image classification component on the `ml/tflite/image-classification` topic. In the AWS IoT console, verify that this subscription exists. If it doesn't, follow the steps in Step 1: Subscribe to the default notifications topic to subscribe to this topic before you deploy the component to your Greengrass core device.

Step 3: Modify the TensorFlow Lite image classification component configuration and deploy it

In this step, you configure and deploy the TensorFlow Lite image classification component to your core device:

To configure and deploy the TensorFlow Lite image classification component (console)

1. In the AWS IoT Greengrass console navigation menu, choose **Components**.
2. On the **Components** page, on the **Public components** tab, choose `aws.greengrass.TensorFlowLiteImageClassification`.
3. On the **aws.greengrass.TensorFlowLiteImageClassification** page, choose **Deploy**.
4. From **Add to deployment**, choose one of the following:

   a. To merge this component to an existing deployment on your target device, choose **Add to existing deployment**, and then select the deployment that you want to revise.

   b. To create a new deployment on your target device, choose **Create new deployment**. If you have an existing deployment on your device, choosing this step replaces the existing deployment.
5. On the **Specify target** page, do the following:

a. Under **Deployment** information, enter or modify the friendly name for your deployment.

b. Under **Deployment targets**, select a target for your deployment, and choose **Next**. You cannot change the deployment target if you are revising an existing deployment.

6. On the **Select components** page, under **Public components**, verify that the `aws.greengrass.TensorFlowLiteImageClassification` component is selected, and choose **Next**.

7. On the **Configure components** page, do the following:

   a. Select the inference component, and choose **Configure component**.

   b. Under **Configuration update**, enter the following configuration update in the **Configuration to merge** box.

   ```
   {
     "InferenceInterval": "60",
     "UseCamera": "true"
   }
   ```

   With this configuration update, the component accesses the camera module on your device and performs inference on images taken by the camera. The inference code runs every 60 seconds.

   c. Choose **Confirm**, and then choose **Next**.

8. On the **Configure advanced settings** page, keep the default configuration settings, and choose **Next**.

9. On the **Review** page, choose **Deploy**

To configure and deploy the TensorFlow Lite image classification component (AWS CLI)

1. Create a `deployment.json` file to define the deployment configuration for the TensorFlow Lite image classification component. This file should look like the following:

   ```
   {
     "targetArn":"targetArn",
     "components": {
       "aws.greengrass.TensorFlowLiteImageClassification": {
         "componentVersion": 2.1.0,
         "configurationUpdate": {
   ```

```
        "InferenceInterval": "60",
        "UseCamera": "true"
      }
    }
  }
}
```

- In the `targetArn` field, replace *`targetArn`* with the Amazon Resource Name (ARN) of the thing or thing group to target for the deployment, in the following format:
  - Thing: `arn:aws:iot:`*`region`*`:`*`account-id`*`:thing/`*`thingName`*
  - Thing group: `arn:aws:iot:`*`region`*`:`*`account-id`*`:thinggroup/`*`thingGroupName`*
- This tutorial uses component version 2.1.0. In the `aws.greengrass.TensorFlowLiteImageClassification` component object, replace *`2.1.0`* to use a different version of the TensorFlow Lite image classification component.

With this configuration update, the component accesses the camera module on your device and performs inference on images taken by the camera. The inference code runs every 60 seconds. Replace the following values

2. Run the following command to deploy the TensorFlow Lite image classification component on the device:

```
aws greengrassv2 create-deployment \
    --cli-input-json file://path/to/deployment.json
```

The deployment can take several minutes to complete. In the next step, check the component log to verify that the deployment completed successfully and to view the inference results.

Step 4: View inference results

After you deploy the component, you can view the inference results in the component log on your Greengrass core device and in the AWS IoT MQTT client in the AWS IoT console. To subscribe to the topic on which the component publishes inference results, see Step 2: Verify your subscription to the default notifications topic.

- **AWS IoT MQTT client**—To view the results that the inference component publishes on the default notifications topic, complete the following steps:

1. In the AWS IoT console navigation menu, choose **Test, MQTT test client**.
2. Under **Subscriptions**, choose **`ml/tflite/image-classification`**.

- **Component log**—To view the inference results in the component log, run the following command on your Greengrass core device.

```
sudo tail -f /greengrass/v2/logs/aws.greengrass.TensorFlowLiteImageClassification.log
```

If you can't see inference results in the component log or in the MQTT client, the deployment failed or didn't reach the core device. This can occur if your core device isn't connected to the internet or doesn't have the required permissions to run the component. Run the following command on your core device to view the AWS IoT Greengrass Core software log file. This file includes logs from the Greengrass core device's deployment service.

```
sudo tail -f /greengrass/v2/logs/greengrass.log
```

For more information, see Troubleshooting machine learning inference.

Next steps

This tutorial shows you how to use the TensorFlow Lite image classification component, with custom configuration options to perform sample image classification on images taken by a camera.

For more information about customizing the configuration of public components or creating custom machine learning components, see Customize your machine learning components.

Tutorial: Set up high availability for AWS IoT Greengrass V2 with Pacemaker

This tutorial shows you how to set up AWS AWS IoT AWS IoT Greengrass V2 in a high availability (HA) configuration using Pacemaker, a high-availability cluster resource manager. You configure multiple Amazon EC2 instances as cluster instances, use DRBD for block-level data replication, and manage AWS IoT Greengrass V2 and load balancer services as Pacemaker resources with automatic failover.

Important

This tutorial uses Amazon Elastic Compute Cloud instances to demonstrate the setup. You can deploy the AWS IoT Greengrass V2 and Pacemaker integration to achieve high availability on a cluster of any device type, as long as the devices can communicate with one another.

This tutorial includes the following setups:

- **Active/Passive AWS IoT Greengrass V2 service** – Run AWS IoT Greengrass V2 as a systemd service managed by Pacemaker with DRBD-replicated storage. Only one instance runs AWS IoT Greengrass V2 at a time, and Pacemaker handles failover to a standby instance.
- **Active/Passive load balancer** – Run HAProxy as a Pacemaker-managed resource with its configuration stored on DRBD-replicated storage. Pacemaker fails over the load balancer to a standby instance if the primary goes down.
- **Active/Active AWS IoT Greengrass V2 component** – Monitor a AWS IoT Greengrass V2 component across all instances using a custom OCF (Open Cluster Framework) resource agent. Pacemaker detects component failures and triggers recovery without full instance failover.

Each setup is standalone and mutually exclusive. Each setup assumes a fresh start from the prerequisites, with the single DRBD resource repurposed for each setup's needs. Setup 3 (Active/Active) does not use DRBD — skip the DRBD prerequisite steps and install AWS IoT Greengrass V2 to a local path on each instance instead.

In Setups 1 and 2, you create a highly available cluster of AWS IoT Greengrass V2 devices. The cluster contains a *primary instance*, which is the instance that is currently active and running the managed services (such as AWS IoT Greengrass V2 or HAProxy), and one or more *standby instances*, which are idle and waiting to take over if the primary fails. Pacemaker automatically promotes one of the standby instances to primary during failover. In Setup 3 (Active/Active), all instances run the service simultaneously, and Pacemaker handles per-instance recovery rather than failover promotion.

Topics

- Prerequisites and cluster setup
- Active/passive AWS IoT Greengrass V2 service

- Active/passive load balancer
- Active/active AWS IoT Greengrass V2 component

Prerequisites and cluster setup

In this section, you set up the infrastructure for the high availability tutorial. You create a security group, create a key pair, launch Amazon EC2 instances, install Pacemaker and DRBD, create a cluster, and configure block-level replication between instances.

Create and configure a security group

Create a security group that allows all traffic between cluster instances. For this tutorial, you open all ports to simplify setup. This step requires a VPC. If you don't have one, see Create a VPC.

Console

- Open the Amazon EC2 console at https://console.aws.amazon.com/ec2/.
- In the navigation pane, choose **Security Groups**, then choose **Create security group**.
- Enter a name (for example, `pacemaker-cluster-sg`) and a description.
- Select your VPC.
- Choose **Create security group**.
- After creation, select the new security group and edit its inbound rules. Choose **Add rule**. Set **Type** to **All traffic** and **Source** to the security group's own ID. This allows all traffic between instances in this security group.
- Add a second inbound rule for SSH access. Set **Type** to **SSH** and **Source** to **My IP**.

AWS Command Line Interface

- Create the security group.

```
aws ec2 create-security-group \
  --group-name pacemaker-cluster-sg \
  --description "Security group for Pacemaker HA cluster" \
  --vpc-id your-vpc-id
```

  Note the `GroupId` from the output (for example, `sg-0123456789abcdef0`).
- Add an inbound rule to allow all traffic from instances in the same security group.

```
aws ec2 authorize-security-group-ingress \
  --group-id security-group-id \
  --protocol all \
  --source-group security-group-id
```

- Add an inbound rule to allow SSH access from your IP address.

```
aws ec2 authorize-security-group-ingress \
  --group-id security-group-id \
  --protocol tcp \
  --port 22 \
  --cidr your-ip/32
```

Create a key pair

Create a key pair to use for SSH access to all instances. Use the same key pair for all instances in this tutorial.

Console

- Open the Amazon EC2 console at https://console.aws.amazon.com/ec2/.
- In the navigation pane, choose **Key Pairs**, then choose **Create key pair**.
- Enter a name (for example, `pacemaker-key`).
- Select **RSA** as the key pair type and **.pem** as the file format.
- Choose **Create key pair**. The private key file downloads automatically.

> **Warning**
>
> Save the private key file (`.pem`) in a secure location. You cannot download it again. You need this file to SSH into your instances.

AWS Command Line Interface

- Create the key pair.

```
aws ec2 create-key-pair \
```

```
  --key-name key-name \
  --query 'KeyMaterial' \
  --output text > pacemaker-key.pem
```

- Set the correct permissions on the private key file.

```
chmod 400 pacemaker-key.pem
```

Launch EC2 instances

Launch two Ubuntu Amazon EC2 instances in the security group you created. A small instance type (for example, `t3.small`) is sufficient for this tutorial.

Console

- Open the Amazon EC2 console at https://console.aws.amazon.com/ec2/.
- Choose **Launch instances**.
- Enter a name for the instance (for example, `pacemaker-instance-1`).
- Under **Application and OS Images**, search for **Ubuntu** and select an Ubuntu Server AMI (for example, Ubuntu Server 24.04 LTS).
- Under **Instance type**, select **t3.small**.
- Under **Key pair**, select the key you created earlier.
- Under **Network settings**, choose **Edit**, select your VPC and a subnet, and under **Firewall (security groups)**, choose **Select existing security group** and select `pacemaker-cluster-sg`.
- Choose **Launch instance**.
- Repeat for each additional instance (for example, `pacemaker-instance-2`). Ensure all instances are in the same VPC and security group.

AWS Command Line Interface

- Launch an instance.

```
aws ec2 run-instances \
  --image-id ubuntu-ami-id \
  --instance-type t3.small \
  --key-name key-name \
  --security-group-ids security-group-id \
```

```
  --subnet-id subnet-id \
  --count 1 \
  --tag-specifications 'ResourceType=instance,Tags=[{Key=Name,Value=pacemaker-
instance-1}]'
```

Repeat with `Value=pacemaker-instance-2` for the second instance.

You need the private IP address of each instance for cluster configuration.

- **Console:** In the Amazon EC2 console, select an instance and find the **Private IPv4 address** in the **Details** tab.
- **AWS Command Line Interface:**

```
aws ec2 describe-instances \
  --filters "Name=tag:Name,Values=pacemaker-instance-*" \
  --query "Reservations[*].Instances[*].[Tags[?Key=='Name'].Value|
[0],PrivateIpAddress]" \
  --output table
```

Install Pacemaker utilities

SSH into each instance and install Pacemaker utilities.

```
ssh -i pacemaker-key.pem ubuntu@instance-public-ip
```

```
sudo apt update
sudo apt install -y pacemaker pcs resource-agents-extra resource-agents-common
 resource-agents-base
```

Configure network communication

1. **Test communication between instances.** On the destination instance, run the following command to listen for pings from the source instance.

   ```
   sudo tcpdump host source-instance-private-ip
   ```

   On the source instance, run the following command to ping the destination instance.

```
ping -c 4 destination-instance-private-ip
```

You should see ping responses on both sides. If not, verify that your security group allows all traffic between instances.

2. **Disable cloud-init hostname management.** On Ubuntu Amazon EC2 instances, cloud-init can revert hostname and `/etc/hosts` changes on reboot. Run this on each instance to prevent that.

```
sudo bash -c 'cat > /etc/cloud/cloud.cfg.d/99-preserve-hostconfig.cfg <<EOF
preserve_hostname: true
manage_etc_hosts: false
EOF'
```

3. **Set the hostname on each instance.** Pacemaker uses hostnames to identify cluster instances.

   On instance 1:

```
sudo hostnamectl set-hostname instance1
```

   On instance 2:

```
sudo hostnamectl set-hostname instance2
```

4. **Configure `/etc/hosts`.** On each instance, append the private IP and hostname pairs to `/etc/hosts`. This allows instances to resolve each other by hostname.

```
sudo tee -a /etc/hosts <<EOF
instance1-private-ip instance1
instance2-private-ip instance2
EOF
```

Create the Pacemaker cluster

1. **Start the pcs daemon** on each instance.

```
sudo systemctl enable pcsd.service
sudo systemctl start pcsd.service
```

2. **Set the hacluster password.** The `hacluster` user is a system account created automatically when you install the `pcs` package. Pacemaker uses this account for cluster authentication between instances. Set the same password on each instance.

```
sudo passwd hacluster
```

3. **Authorize pcs hosts.** Authorize the cluster instances with each other. Run this on each instance.

```
sudo pcs host auth instance1 instance2
```

   You will be prompted for the `hacluster` username and password.

4. **Set up the cluster.** Run this command on one instance only. The `--force` flag destroys any existing cluster on the device.

```
sudo pcs cluster setup my-gg-cluster instance1 instance2 --force
```

5. **Start the cluster.** Run this command on one instance only.

```
sudo pcs cluster start --all
sudo pcs cluster enable --all
```

6. **Verify the cluster status** on either instance.

```
sudo pcs status
```

Set up DRBD

DRBD (Distributed Replicated Block Device) is a Linux kernel module that replicates storage devices between instances in real time. It works at the block device level — the raw disk level below the filesystem — so all data written to disk on one instance is mirrored to the other. This ensures that if the primary instance fails, the standby instance has an identical copy of the data.

1. **Create and attach EBS volumes.** Create an EBS volume for each instance. The volume must be in the same Availability Zone as the instance.

   Console

   - Open the Amazon EC2 console at https://console.aws.amazon.com/ec2/.
   - In the navigation pane, choose **Volumes**, then choose **Create volume**.

- Set **Volume type** to **gp3**, **Size** to **8 GiB**, and **Availability Zone** to the same zone as your instance.
- Choose **Create volume**.
- Select the new volume, choose **Actions** > **Attach volume**, select the target instance, set the device name to `/dev/xvdbz`, and choose **Attach volume**.
- Repeat for each instance.

AWS Command Line Interface

- Create a volume in the same Availability Zone as your instance.

```
aws ec2 create-volume \
  --volume-type gp3 \
  --size 8 \
  --availability-zone instance-availability-zone
```

Note the `VolumeId` from the output.

- Attach the volume to an instance.

```
aws ec2 attach-volume \
  --volume-id volume-id \
  --instance-id instance-id \
  --device /dev/xvdbz
```

Repeat for each instance.

2. **Verify the volume.** SSH into each instance and verify the volume is attached by listing block devices.

```
lsblk
```

On Nitro-based instance types such as `t3.small`, the volume appears as an NVMe device (for example, `/dev/nvme1n1`) instead of `/dev/xvdbz`. Use `lsblk` to identify the actual device name, and use that name in the DRBD configuration.

3. **Install DRBD utilities** on each instance.

```
sudo apt install -y drbd-utils
```

4. **Configure the DRBD resource.** Create the DRBD resource configuration directory and file on both instances.

```
sudo mkdir -p /etc/drbd.d
sudo vi /etc/drbd.d/greengrass.res
```

   Add the following configuration. Replace the hostnames and IP addresses with your values.

```
resource greengrass {
    protocol C;

    on instance1 {
        device    /dev/drbd0;
        disk      /dev/instance1-device-name;
        address   instance1-private-ip:7788;
        meta-disk internal;
    }

    on instance2 {
        device    /dev/drbd0;
        disk      /dev/instance2-device-name;
        address   instance2-private-ip:7788;
        meta-disk internal;
    }
}
```

5. **Create and start the DRBD resource.** Run on both instances.

```
sudo drbdadm create-md greengrass
sudo drbdadm up greengrass
```

   Verify with `lsblk` that `drbd0` appears under your EBS volume device (for example, `xvdbz` or `nvme1n1`).

6. **Set the primary instance.** Run the following commands on instance1 only to force it to be primary and wait for synchronization.

```
sudo drbdadm -- --overwrite-data-of-peer primary greengrass
sudo drbdadm status
```

Wait until both instances show `UpToDate` for the disk state. Once synchronized, all other instances become standby instances.

7. **Format and mount the DRBD device.**

   On the primary instance only, format the DRBD device:

   ```
   sudo mkfs.ext4 /dev/drbd0
   ```

   On all instances, create the mount directory:

   ```
   sudo mkdir -p /greengrass/v2
   ```

   On the primary instance only, mount the DRBD device:

   ```
   sudo mount /dev/drbd0 /greengrass/v2
   ```

8. **Verify DRBD sync status.**

   ```
   sudo drbdadm status
   mount | grep 'drbd'
   ```

Active/passive AWS IoT Greengrass V2 service

In this setup, you run AWS IoT Greengrass V2 as a systemd service on one instance at a time. Pacemaker manages the DRBD replication, filesystem mount, and AWS IoT Greengrass V2 service as ordered resources. If the primary instance fails, Pacemaker promotes the standby instance's DRBD to primary, mounts the filesystem, and starts AWS IoT Greengrass V2.

> **Important**
>
> Complete all steps in the section called “Prerequisites and cluster setup” before proceeding.

> **Warning**
>
> Run the following commands on the primary instance only, unless otherwise noted.

Attach the DRBD resource

Verify that Pacemaker is running before proceeding.

```
sudo systemctl status pacemaker
```

Disable STONITH before creating any resources. Without a fencing device configured, Pacemaker will refuse to start resources if STONITH is enabled (the default).

```
sudo pcs property set stonith-enabled=false
```

> **Warning**
>
> STONITH is disabled here to simplify this tutorial. In a production environment, you must enable STONITH and configure a fencing agent (for example, `fence_aws` for Amazon EC2 instances) to prevent split-brain and data corruption.

Unmount the DRBD device on the primary instance and bring DRBD down on all instances so that Pacemaker has clean control of the DRBD lifecycle.

```
# On the primary instance only
sudo umount /greengrass/v2

# On all instances
sudo drbdadm down greengrass
```

Create the DRBD resource in Pacemaker.

```
sudo pcs resource create drbd-greengrass \
  ocf:linbit:drbd drbd_resource=greengrass \
  op monitor interval=15s role=Promoted \
  op monitor interval=30s role=Unpromoted
```

Configure the resource as promotable so that only one instance is primary at a time. Set `clone-max` to the number of instances in your cluster.

```
sudo pcs resource promotable drbd-greengrass \
  promoted-max=1 promoted-node-max=1 clone-max=2 clone-node-max=1 notify=true
```

Attach the filesystem resource

All AWS IoT Greengrass V2 resources are stored under `/greengrass/v2`. This step tells Pacemaker to mount the DRBD device at that path on the promoted instance. This ensures the AWS IoT Greengrass V2 data directory is replicated and available during failover.

Create the filesystem resource in a disabled state. You will enable it after all constraints are in place.

```
sudo pcs resource create fs_greengrass Filesystem \
    device="/dev/drbd0" \
    directory="/greengrass/v2" \
    fstype="ext4" \
    op start timeout=15s \
    op stop timeout=15s \
    --disabled
```

Verify resources

Verify that the resources are created and propagated to all instances.

```
sudo pcs status
```

Provision and attach the AWS IoT Greengrass V2 systemd resource

1. **Provision AWS IoT Greengrass V2.** Because you unmounted the DRBD device earlier, enable maintenance mode and remount it for provisioning:

   ```
   # Enable maintenance mode to prevent Pacemaker from interfering
   sudo pcs property set maintenance-mode=true

   # Check which node Pacemaker promoted to Primary
   sudo pcs status | grep drbd-greengrass

   # On the Promoted node, mount the DRBD device
   sudo mount /dev/drbd0 /greengrass/v2
   ```

   Provision AWS IoT Greengrass V2 on the primary instance using automatic provisioning. Follow the instructions in Install AWS IoT Greengrass V2 Core software with automatic resource provisioning.

Ensure that AWS IoT Greengrass V2 is installed to the `/greengrass/v2` directory (the DRBD-mounted path). After provisioning, unmount the device and disable maintenance mode:

```
sudo umount /greengrass/v2
sudo pcs property set maintenance-mode=false
```

2. **Disable the AWS IoT Greengrass V2 service** so that Pacemaker can manage it instead of systemd.

```
sudo systemctl disable greengrass
sudo systemctl stop greengrass
```

3. **Install runtime prerequisites on all standby instances.** AWS IoT Greengrass V2 requires Java and other dependencies that the automatic provisioner installs outside the DRBD-replicated directory. Install the same JDK version on each standby instance. See AWS IoT Greengrass V2 requirements for the full list of prerequisites.

4. **Create systemd unit file on other instances.** Copy the AWS IoT Greengrass V2 systemd unit file to the standby instances so that Pacemaker can start the service on any instance during failover.

```
# On the primary instance, view the unit file location
systemctl show -p FragmentPath greengrass
```

Copy that file to the same path on each other instance. Then reload systemd on each standby instance so it recognizes the new unit file.

```
# On each standby instance
sudo systemctl daemon-reload
```

5. **Attach the AWS IoT Greengrass V2 resource.**

```
sudo pcs resource create greengrass systemd:greengrass \
  op monitor interval=10s \
  op start timeout=60s \
  op stop timeout=60s \
  --disabled
```

Create resource constraints

Create constraint ordering so that Pacemaker manages the resources in the correct order during failover.

```
sudo pcs constraint order promote drbd-greengrass-clone then start fs_greengrass
sudo pcs constraint colocation add fs_greengrass with Promoted drbd-greengrass-clone
 score=INFINITY
sudo pcs resource group add greengrass-group fs_greengrass greengrass
sudo pcs constraint location greengrass-group prefers instance1=200
```

Enable the resources now that constraints are in place.

```
sudo pcs resource enable fs_greengrass
sudo pcs resource enable greengrass
```

Verify the final state of the resource constraints.

```
sudo pcs constraint show
```

The output should show the following constraints:

- **Location Constraints** – The `greengrass-group` resource group prefers the primary instance.
- **Colocation Constraints** – `fs_greengrass` runs with the promoted `drbd-greengrass-clone`, and `greengrass` runs with `fs_greengrass`.
- **Order Constraints** – DRBD promotes before the filesystem starts, and the filesystem starts before AWS IoT Greengrass V2.

Verify failover

Simulate a failover to verify that the setup works.

1. **Check the initial state.** Verify that AWS IoT Greengrass V2 is running on the primary instance.

   ```
   sudo pcs status
   ```

2. **Simulate primary instance failure.** Put the primary node in standby mode to trigger resource migration.

```
sudo pcs node standby primary-node-name
```

3. **Verify failover.** On the standby instance, check the cluster status. The DRBD, filesystem, and AWS IoT Greengrass V2 resources should now be running on the standby instance.

   ```
   sudo pcs status
   ```

4. **Recover the failed instance.**

   ```
   sudo pcs node unstandby primary-node-name
   ```

   When the node is brought back online, it rejoins the cluster as a standby instance. The instance that was promoted during failover remains the primary instance.

Troubleshooting

If resources enter a failed state, you can clean up and restart them with the following command.

```
sudo pcs resource cleanup
```

Active/passive load balancer

In this setup, you run HAProxy as a load balancer on one instance at a time. The HAProxy configuration files are stored on DRBD-replicated storage so that when Pacemaker fails over to a standby instance, the configuration is available immediately.

Important

Complete all steps in the section called “Prerequisites and cluster setup” before proceeding.

Warning

Run the following commands on the primary instance only, unless otherwise noted.

Install HAProxy on all instances

Install HAProxy and disable the service so that Pacemaker can manage it. Run this on each instance.

```
sudo apt install -y haproxy
sudo systemctl disable haproxy
sudo systemctl stop haproxy
```

Configure the DRBD mount for HAProxy

Mount the DRBD device to a directory for the load balancer.

Create the mount point directory. Run this on each instance.

```
sudo mkdir -p /drbd/loadbalancer/
```

Mount the DRBD device on the primary instance. First unmount the previous mount point from the prerequisites.

```
sudo umount /greengrass/v2
sudo mount /dev/drbd0 /drbd/loadbalancer/
```

Verify the mount with `lsblk`. You should see `drbd0` mounted at `/drbd/loadbalancer`.

Configure HAProxy

The configuration files must be on the DRBD mount directory so they are replicated to the failover instance.

1. **Copy the HAProxy configuration** to the DRBD mount directory.

   ```
   sudo mkdir -p /drbd/loadbalancer/etc/haproxy/
   sudo cp /etc/haproxy/haproxy.cfg /drbd/loadbalancer/etc/haproxy/haproxy.cfg
   ```

2. **Edit the HAProxy systemd unit file** to use the configuration from the DRBD mount path. Run this on each instance.

   ```
   sudo systemctl edit haproxy
   ```

Add the following lines to update the configuration file path to the DRBD mount path.

```
[Service]
Environment="CONFIG=/drbd/loadbalancer/etc/haproxy/haproxy.cfg"
```

3. **Reload systemd.** Run this on each instance.

```
sudo systemctl daemon-reload
```

Attach the DRBD resource

Unmount the DRBD device and bring DRBD down so that Pacemaker can manage it. Run the unmount on the primary instance and `drbdadm down` on all instances.

```
# On the primary instance only
sudo umount /drbd/loadbalancer

# On all instances
sudo drbdadm down greengrass
```

```
sudo pcs resource create drbd-greengrass \
  ocf:linbit:drbd drbd_resource=greengrass \
  op monitor interval=15s role=Promoted \
  op monitor interval=30s role=Unpromoted
```

```
sudo pcs resource promotable drbd-greengrass \
  promoted-max=1 promoted-node-max=1 clone-max=2 clone-node-max=1 notify=true
```

Attach the filesystem resource

```
sudo pcs resource create fs_loadbalancer Filesystem \
    device="/dev/drbd0" \
    directory="/drbd/loadbalancer" \
    fstype="ext4" \
    op start timeout=15s \
    op stop timeout=15s \
    --disabled
```

Attach the HAProxy systemd resource

```
sudo pcs resource create haproxy systemd:haproxy \
  op monitor interval=10s \
  op start timeout=60s \
  op stop timeout=60s \
  --disabled
```

Create resource constraints

```
sudo pcs constraint colocation add haproxy with fs_loadbalancer
sudo pcs constraint order fs_loadbalancer then start haproxy
sudo pcs constraint colocation add fs_loadbalancer with Promoted drbd-greengrass-clone
sudo pcs constraint order promote drbd-greengrass-clone then start fs_loadbalancer
```

Enable the resources now that constraints are in place.

```
sudo pcs resource enable fs_loadbalancer
sudo pcs resource enable haproxy
```

Disable STONITH for this tutorial setup.

```
sudo pcs property set stonith-enabled=false
```

> **Warning**
>
> STONITH is disabled here to simplify this tutorial. In a production environment, you must enable STONITH and configure a fencing agent (for example, `fence_aws` for Amazon EC2 instances) to prevent split-brain and data corruption.

Verify failover

1. **Check the initial state.** Verify that HAProxy is running on the primary instance.

   ```
   sudo pcs status
   ```

2. **Simulate failover.** Put the primary node in standby mode to force all resources off the primary node.

```
sudo pcs node standby primary-node-name
```

3. **Verify failover.** On the standby instance, check the cluster status. The DRBD, filesystem, and HAProxy resources should now be running on the standby instance with up-to-date configuration files.

   ```
   sudo pcs status
   ```

   When the failed instance recovers, the load balancer service remains on the standby instance unless you configure Pacemaker to migrate it.

4. **Bring the node back online.**

   ```
   sudo pcs node unstandby primary-node-name
   ```

Active/active AWS IoT Greengrass V2 component

In this setup, you manage a AWS IoT Greengrass V2 component with Pacemaker using a custom OCF (Open Cluster Framework) resource agent. This allows Pacemaker to monitor the health of a AWS IoT Greengrass V2 component and trigger recovery actions when the component enters a broken state.

> **Important**
>
> Complete all steps in the section called "Prerequisites and cluster setup" before proceeding, except for the DRBD setup steps. This setup does not use DRBD. Install AWS IoT Greengrass V2 to a local path on each instance instead. AWS IoT Greengrass V2 must be provisioned and running on all instances. This tutorial assumes AWS IoT Greengrass V2 is installed at `/greengrass/v2`. If you chose a different path, update the `GG_CLI` variable in the OCF script accordingly.

Create a custom OCF resource agent

Create the custom resource agent directory and script on all instances. This example manages a component named `PythonWebServer`.

```
sudo mkdir -p /usr/lib/ocf/resource.d/custom
```

Create the resource agent script at `/usr/lib/ocf/resource.d/custom/gg-webserver` with the following content.

```
#!/bin/bash
# OCF Resource Agent for Greengrass Web Server component
. /usr/lib/ocf/lib/heartbeat/ocf-shellfuncs

GG_CLI="/greengrass/v2/bin/greengrass-cli"
COMPONENT="PythonWebServer"
STATE_FILE="/run/gg-webserver.ocf-state"

case "$1" in
  meta-data)
  cat <<EOF
<?xml version="1.0"?>
<resource-agent name="gg-webserver">
  <version>1.0</version>
  <longdesc lang="en">Greengrass webserver component agent</longdesc>
  <shortdesc lang="en">GG Webserver</shortdesc>
  <parameters>
  </parameters>
  <actions>
    <action name="start" timeout="60"/>
    <action name="stop" timeout="10"/>
    <action name="monitor" timeout="5" interval="10"/>
    <action name="meta-data" timeout="5"/>
  </actions>
</resource-agent>
EOF
;;
  start)
    touch "$STATE_FILE"
    systemctl restart greengrass
    if [ $? -eq 0 ]; then
      exit $OCF_SUCCESS
    else
      rm -f "$STATE_FILE"
      exit $OCF_ERR_GENERIC
    fi
    ;;
  stop)
```

```
    rm -f "$STATE_FILE"
    exit $OCF_SUCCESS
    ;;
  monitor)
    # Check state file first — if absent, resource is stopped
    [ ! -f "$STATE_FILE" ] && exit $OCF_NOT_RUNNING
    # Check if the Greengrass service is running
    if ! systemctl is-active --quiet greengrass; then
      exit $OCF_NOT_RUNNING
    fi
    STATE=$($GG_CLI component details -n=$COMPONENT 2>/dev/null | grep
 '^[[:space:]]*State:' | awk '{print $2}')
    if [[ -z "$STATE" ]]; then
      ocf_log warn "Component $COMPONENT state is empty — component may not be
 deployed"
      exit $OCF_SUCCESS
    elif [[ "$STATE" == "BROKEN" ]]; then
      exit $OCF_ERR_GENERIC
    else
      exit $OCF_SUCCESS
    fi
    ;;
  *)
    echo "Usage: $0 {start|stop|monitor|meta-data}"
    exit $OCF_ERR_UNIMPLEMENTED
    ;;
esac
```

Make the script executable.

```
sudo chmod +x /usr/lib/ocf/resource.d/custom/gg-webserver
```

Note

The `start` action restarts the entire AWS IoT Greengrass V2 service, which restarts all components on the instance, not just `PythonWebServer`. This is the only practical recovery path because AWS IoT Greengrass V2 does not support restarting individual components. The `stop` action is intentionally a no-op because this agent is a monitoring wrapper — the AWS IoT Greengrass V2 service lifecycle is managed by systemd, not by this agent. If a component remains persistently `BROKEN` (for example, due to a bad deployment), Pacemaker will retry up to `migration-threshold` times, then ban the

resource on that node until `failure-timeout` expires. You must fix the root cause (for example, redeploy a valid component version) to stop the retry cycle.

Attach the resource

Create the Pacemaker resource using the custom OCF agent.

```
sudo pcs property set stonith-enabled=false
```

Warning

STONITH is disabled here to simplify this tutorial. In a production environment, you must enable STONITH and configure a fencing agent (for example, `fence_aws` for Amazon EC2 instances) to prevent split-brain and data corruption.

```
sudo pcs resource create gg-webserver ocf:custom:gg-webserver \
  op monitor interval=30s \
  op start timeout=60s \
  meta migration-threshold=3 failure-timeout=60s \
  clone
```

Verify recovery

1. **Check the initial state.** Verify that the AWS IoT Greengrass V2 component is running and healthy on all instances.

   ```
   sudo pcs status
   ```

2. **Simulate component failure.** Kill the component's process to simulate a transient failure. AWS IoT Greengrass V2 might attempt internal recovery first. If the component enters a BROKEN state, Pacemaker detects it and triggers a service restart. If AWS IoT Greengrass V2 recovers the component internally, Pacemaker takes no action.

   ```
   sudo pkill -f "PythonWebServer"

   # Wait 30-60 seconds, then check the component state
   ```

```
sudo /greengrass/v2/bin/greengrass-cli component details -n=PythonWebServer
```

3. **Verify recovery.** Pacemaker detects that the component is unhealthy and performs recovery steps as defined in the custom OCF script. No failover is needed — Pacemaker restarts the service on the same instance.

   Other services such as HAProxy and AWS IoT Greengrass V2 continue to operate normally on all instances. The application on the standby instances continues to take requests without interruption.

   ```
   sudo pcs status
   ```

   When the recovered instance comes back up, the load balancer identifies it and distributes client requests as needed.

Switch a Greengrass core device to a different AWS account or Region

You can deploy a nucleus configuration update to switch a core device's IoT connectivity from one AWS account or Region to another. This is useful in the following scenarios:

- **Manufacturing to production transition** – You provision devices in a staging account during manufacturing and move them to the destination account during field installation.
- **Geographic relocation** – Devices that move from one Region to another must connect to the local Region's AWS IoT endpoint to reduce latency.
- **Failure impact reduction** – You redistribute devices initially concentrated in a single account across multiple accounts to limit the impact of account-level issues such as service quota limits or security incidents.

When you deploy an IoT connectivity switch, the nucleus validates connectivity to the new endpoint before applying the change and automatically rolls back if the connection fails.

This feature requires Greengrass nucleus v2.18.0 or later, or Greengrass nucleus lite v2.6.0 or later.

Topics

- Prerequisites

- Create the deployment
- What to expect
- Verify the switch

Prerequisites

Before you deploy a connectivity switch, set up the following in the destination AWS account and Region:

- **AWS IoT thing** – Create a thing with the same name as in the source account.
- **Certificate** – Register the same device certificate in the destination account. The certificate must be the same certificate that the device uses in the source account.
- **AWS IoT policy** – Attach a policy to the certificate. The policy must allow `iot:Connect` with a resource ARN that uses a wildcard suffix for the client ID:

  ```
  "Resource": "arn:aws:iot:*:*:client/${iot:Connection.Thing.ThingName}*"
  ```

  You must include the wildcard suffix because the nucleus pre-flight connectivity check uses client ID *`thing-name`*`#endpoint-switch`.
- **AWS IoT role alias** – Create a role alias in the destination Region with the appropriate IAM role permissions for Greengrass operations.

> **Note**
>
> To use AWS IoT thing policy variables such as `${iot:Connection.Thing.ThingName}` in the AWS IoT policy, you must attach the certificate exclusively to the AWS IoT thing in the destination account.

For Greengrass nucleus core devices, you must also set `greengrassDataPlaneEndpoint` to **`iotdata`** in the nucleus configuration. Include this setting in the connectivity switch deployment or during initial provisioning.

Without this setting, the data plane stays bound to the original account after the switch. This breaks component artifact downloads and control plane operations. For more information, see the `greengrassDataPlaneEndpoint` configuration parameter in Greengrass nucleus.

For Greengrass nucleus lite core devices, you must create a private `aws.greengrass.NucleusLite` component in your account before you can deploy configuration updates to the nucleus. AWS IoT Greengrass deployments can only target components that exist in the account's component registry, and there is no public component for nucleus lite.

Create a component with the `aws_nucleus_lite` runtime and an empty lifecycle (no actions). The component version must match the version of nucleus lite running on the device.

Create the deployment

Create a deployment that targets the core device or its thing group in the source account. The deployment sends a merge configuration update to the nucleus component with the new AWS IoT endpoint values.

To deploy a connectivity switch (AWS CLI)

1. Create a file called `switch-deployment.json` with the following content. Replace the placeholder values with your destination account's values.

   - Replace *`targetArn`* with the ARN of the AWS IoT thing group or the core device in the source account.
   - Replace *`componentName`* with `aws.greengrass.Nucleus` for nucleus or `aws.greengrass.NucleusLite` for nucleus lite.
   - Replace *`componentVersion`* with the version of the nucleus running on the device.

```
{
  "targetArn": "targetArn",
  "deploymentName": "SwitchConnectivity",
  "components": {
    "componentName": {
      "componentVersion": "componentVersion",
      "configurationUpdate": {
        "merge": "{\"iotDataEndpoint\":\"account-prefix-
ats.iot.region.amazonaws.com\",\"iotCredEndpoint\":\"account-
prefix.credentials.iot.region.amazonaws.com\",\"awsRegion\":\"region\",
\"iotRoleAlias\":\"role-alias-name\"}"
      }
    }
  },
  "deploymentPolicies": {
```

```
    "failureHandlingPolicy": "ROLLBACK"
  }
}
```

For Greengrass nucleus core devices that do not already have `greengrassDataPlaneEndpoint` set to **`iotdata`**, add it to the merge configuration:

```
"merge": "{\"iotDataEndpoint\":\"...\",\"iotCredEndpoint\":\"...\",\"awsRegion\":
\"...\",\"iotRoleAlias\":\"...\",\"greengrassDataPlaneEndpoint\":\"iotdata\"}"
```

Encode the `merge` value as a JSON string. Escape all inner quotes with backslashes (\"). Use a JSON validation tool to verify syntax before creating the deployment.

2. Run the following command to create the deployment.

```
aws greengrassv2 create-deployment \
  --cli-input-json file://switch-deployment.json
```

What to expect

The nucleus validates connectivity to the new endpoint before applying the change. If the device cannot connect to the new endpoint, the deployment fails and the device remains connected to the source account.

The nucleus uses client ID *`thing-name`*`#endpoint-switch` for the pre-flight connectivity check.

After applying the configuration, the nucleus tests connectivity to the new endpoint. If connectivity fails, the nucleus automatically rolls back to the original configuration. The deployment reports `FAILED` in the source account.

If the connectivity switch succeeds, the deployment reports `SUCCEEDED` in the source account. AWS IoT Greengrass creates a core device in the account and Region associated with the new endpoint.

> **Note**
>
> During the connectivity switch, the nucleus restarts components that have a hard dependency on the token exchange service. This restart refreshes credentials from the new AWS IoT credential endpoint. The nucleus does not restart other components, and they continue running with their existing state.

Verify the switch

After the deployment completes, verify that the connectivity switch succeeded.

To verify the connectivity switch

1. Check the deployment status in the source account. The deployment reports SUCCEEDED after the device connects to the new endpoint.

   ```
   aws greengrassv2 list-effective-deployments \
     --core-device-thing-name thing-name
   ```

2. Verify that the core device appears in the destination account. Open the AWS IoT Greengrass console or run the following command.

   ```
   aws greengrassv2 get-core-device \
     --core-device-thing-name thing-name
   ```

The AWS IoT thing and core device record remain in the source account after the switch. To prevent future deployments from targeting the device, remove it from the source account's thing group.

If the deployment status is not SUCCEEDED, check the Greengrass log at */greengrass/v2*/logs/greengrass.log on the device for error details. For common error codes related to connectivity switch operations, see Detailed deployment error codes.

Components

AWS IoT Greengrass components are software modules that you deploy to Greengrass core devices. Components can represent applications, runtime installers, libraries, or any code that you would run on a device. You can define components that depend on other components. For example, you might define a component that installs Python, and then define that component as a dependency of your components that run Python applications. When you deploy your components to your fleets of devices, Greengrass deploys only the software modules that your devices require.

Topics

- AWS-provided components
- Publisher-supported components
- Community components
- AWS IoT Greengrass development tools
- Develop AWS IoT Greengrass components
- Deploy AWS IoT Greengrass components to devices

AWS-provided components

AWS IoT Greengrass provides and maintains prebuilt components that you can deploy to your devices. These components include features (such as stream manager), AWS IoT Greengrass V1 connectors (such as CloudWatch metrics), and local development tools (such as the AWS IoT Greengrass CLI). You can deploy these components to your devices for their standalone functionality, or you can use them as dependencies in your custom Greengrass components.

> **Note**
>
> Several AWS-provided components depend on specific minor versions of the Greengrass nucleus. Because of this dependency, you need to update these components when you update the Greengrass nucleus to a new minor version. For information about the specific versions of the nucleus that each component depends on, see the corresponding component topic. For more information about updating the nucleus, see Update the AWS IoT Greengrass Core software (OTA).

When a component has a component type of both generic and Lambda, the current version of the component is the generic type and a previous version of the component is the Lambda type.

Component	Description	Componen type	Supported OS	Open source	Nucleus lite support
Greengrass nucleus	The nucleus of the AWS IoT Greengras s Core software. Use this component to configure and update the software on your core devices.	Nucleus	Linux, Windows	Yes	No
Greengrass nucleus lite	A lightweig ht nucleus for resource-constrain ed devices optimized for low-cost, edge devices and high-volu me applicati ons	NucleusLi te	Linux	Yes	No

Component	Description	Componen type	Supported OS	Open source	Nucleus lite support
Client device auth	Enables local IoT devices, called client devices, to connect to the core device.	Plugin	Linux, Windows	Yes	No
CloudWatch metrics	Publishes custom metrics to Amazon CloudWatch.	Generic, Lambda	Linux, Windows	Yes	Yes
AWS IoT Device Defender	Notifies administrators of changes in the state of the Greengrass core device to identify unusual behavior.	Generic, Lambda	Linux, Windows	Yes	No

Component	Description	Componen type	Supported OS	Open source	Nucleus lite support
Disk spooler	Enables a persisten t storage option for messages spooled from Greengrass core devices to AWS IoT Core. This component will store these outbound messages on disk.	Plugin	Linux, Windows	Yes	No
Docker application manager	Enables AWS IoT Greengrass to download Docker images from Docker Hub and Amazon Elastic Container Registry (Amazon ECR).	Generic	Linux, Windows	No	Built-in

Component	Description	Componen type	Supported OS	Open source	Nucleus lite support
Edge connector for Kinesis Video Streams	Reads video feeds from local cameras, publishes the streams to Kinesis Video Streams, and displays the streams in Grafana dashboard s with AWS IoT TwinMaker.	Generic	Linux	No	No
Greengrass CLI	Provides a command-l ine interface that you can use to create local deploymen ts and interact with the Greengrass core device and its component s.	Plugin	Linux, Windows	Yes	No

Component	Description	Componen type	Supported OS	Open source	Nucleus lite support
IP detector	Reports MQTT broker connectivity information to AWS IoT Greengras s, so client devices can discover how to connect.	Plugin	Linux, Windows	Yes	No
Firehose	Publishes data through Amazon Data Firehose delivery streams to destinations in the AWS Cloud.	Lambda	Linux	No	No
Lambda launcher	Handles processes and environme nt configura tion for Lambda functions.	Generic	Linux	No	No

Component	Description	Componen type	Supported OS	Open source	Nucleus lite support
Lambda manager	Handles interprocess communica tion and scaling for Lambda functions.	Plugin	Linux	No	No
Lambda runtimes	Provides artifacts for each Lambda runtime.	Generic	Linux	No	No
Legacy subscription router	Manages subscript ions for Lambda functions that run on AWS IoT Greengrass V1.	Generic	Linux	No	No

Component	Description	Componen type	Supported OS	Open source	Nucleus lite support
Local debug console	Provides a local console that you can use to debug and manage the Greengrass core device and its component s.	Plugin	Linux, Windows	Yes	No
Log manager	Collects and uploads logs on the Greengrass core device.	Plugin	Linux, Windows	Yes	No
Machine learning components	Provides machine learning models and sample inference code that you can use to perform machine learning inference on Greengrass core devices.	See Machine learning component s.			No

Component	Description	Component type	Supported OS	Open source	Nucleus lite support
Modbus-RTU protocol adapter	Polls information from local Modbus RTU devices.	Lambda	Linux	No	No
Nucleus telemetry emitter	Publishes system health telemetry data gathered from the nucleus to a local topic or to an AWS IoT Core MQTT topic.	Plugin	Linux, Windows	Yes	No
MQTT bridge	Relays MQTT messages between client devices, local AWS IoT Greengrass publish/subscribe, and AWS IoT Core.	Plugin	Linux, Windows	Yes	No

Component	Description	Componen type	Supported OS	Open source	Nucleus lite support
MQTT 3.1.1 broker (Moquette)	Runs an MQTT 3.1.1 broker that handles messages between client devices and the core device.	Plugin	Linux, Windows	Yes	No
MQTT 5 broker (EMQX)	Runs an MQTT 5 broker that handles messages between client devices and the core device.	Generic	Linux, Windows	No	No

Component	Description	Componen type	Supported OS	Open source	Nucleus lite support
PKCS#11 provider	Enables Greengrass components to to access a private key and certificate that you securely store in a hardware security module (HSM).	Plugin	Linux	Yes	Built-in
Secret manager	Deploys secrets from AWS Secrets Manager secrets so that you can securely use credentia ls, such as passwords , in custom component s on the Greengrass core device.	Plugin	Linux, Windows	Yes	No

Component	Description	Componen type	Supported OS	Open source	Nucleus lite support
Secure tunneling	Enables AWS IoT secure tunneling connectio ns that you can use to establish bidirectional communica tions with Greengrass core devices that are behind restricted firewalls.	Generic	Linux	No	Yes

Component	Description	Componen type	Supported OS	Open source	Nucleus lite support
Shadow manager	Enables interacti on with shadows on the core device. It manages shadow document storage and also the synchroni zation of local shadow states with the AWS IoT Device Shadow service.	Plugin	Linux, Windows	Yes	No
Amazon SNS	Publishes messages to Amazon SNS topics.	Lambda	Linux	No	No
Stream manager	Streams high-volu me data from local sources to the AWS Cloud.	Generic	Linux, Windows	No	Yes

Component	Description	Componen type	Supported OS	Open source	Nucleus lite support
System log forwarder	Upload systemd-j ournald logs to the AWS Cloud.	Generic	Linux	Yes	Yes
Systems Manager Agent	Manage the core device with AWS Systems Manager, which enables you to patch devices, run commands, and more.	Generic	Linux	Yes	No
Token exchange service	Provides AWS credentia ls that you can use to interact with AWS services.	Generic	Linux, Windows	No	Built-in
IoT SiteWise OPC UA collector	Collects data from OPC-UA servers.	Generic	Linux, Windows	No	No

Component	Description	Componen type	Supported OS	Open source	Nucleus lite support
IoT SiteWise OPC UA data source simulator	Runs a local OPC-UA server that generates sample data.	Generic	Linux, Windows	No	No
IoT SiteWise publisher	Publishes data to the AWS Cloud.	Generic	Linux, Windows	No	No
IoT SiteWise processor	Processes data on the Greengrass core devices.	Generic	Linux, Windows	No	No

Greengrass nucleus

The Greengrass nucleus component (`aws.greengrass.Nucleus`) is a mandatory component and the minimum requirement to run the AWS IoT Greengrass Core software on a device. You can configure this component to customize and update your AWS IoT Greengrass Core software remotely. Deploy this component to configure settings such as proxy, device role, and AWS IoT thing configuration on your core devices.

> **Note**
>
> As of Greengrass version 2.14.0, a memory footprint optimized version of the nucleus device runtime is available for constrained edge devices. See Greengrass nucleus lite for more information on its configuration and use.

Important

When the version of the nucleus component changes, or when you change certain configuration parameters, the AWS IoT Greengrass Core software—which includes the nucleus and all other components on your device—restarts to apply the changes.
When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

Topics

- Versions
- Device requirements
- Supported platforms
- Operating system
- Requirements
- Dependencies
- Download and installation
- Configuration
- Local log file
- Changelog

Versions

This component has the following versions:

- 2.18.x
- 2.17.x

- 2.16.x
- 2.15.x
- 2.14.x
- 2.13.x
- 2.12.x
- 2.11.x
- 2.10.x
- 2.9.x
- 2.8.x
- 2.7.x
- 2.6.x
- 2.5.x
- 2.4.x
- 2.3.x
- 2.2.x
- 2.1.x
- 2.0.x

Device requirements

> **Note**
>
> You can use AWS IoT Device Tester for AWS IoT Greengrass to verify that your device can run the AWS IoT Greengrass Core software and communicate with the AWS Cloud. For more information, see Using AWS IoT Device Tester for AWS IoT Greengrass V2.

Linux

- The use of an AWS Region that supports AWS IoT Greengrass V2. For the list of supported Regions, see AWS IoT Greengrass V2 endpoints and quotas in the *AWS General Reference*.
- Minimum 256 MB disk space available for the AWS IoT Greengrass Core software. This requirement doesn't include components deployed to the core device.

- Minimum 96 MB RAM allocated to the AWS IoT Greengrass Core software. This requirement doesn't include components that run on the core device. For more information, see Control memory allocation with JVM options.
- Java Runtime Environment (JRE) version 8 or greater. Java must be available on the PATH environment variable on the device. To use Java to develop custom components, you must install a Java Development Kit (JDK). We recommend that you use Amazon Corretto or OpenJDK long-term support versions. Version 8 or higher is required.
- GNU C Library (glibc) version 2.25 or greater.
- You can run the AWS IoT Greengrass Core software as a root user. Use `sudo`, for example. You can also run the AWS IoT Greengrass Core software as a non-root user. For more information, see Set up AWS IoT Greengrass V2 core devices as non-root.
- The root user that runs the AWS IoT Greengrass Core software, such as `root`, must have permission to run `sudo` with any user and any group. The `/etc/sudoers` file must give this user permission to run `sudo` as other groups. The permission for the user in `/etc/sudoers` should look like the following example.

  ```
  root    ALL=(ALL:ALL) ALL
  ```

- The core device must be able to perform outbound requests to a set of endpoints and ports. For more information, see Allow device traffic through a proxy or firewall.
- The `/tmp` directory must be mounted with `exec` permissions.
- All of the following shell commands:
  - `ps -ax -o pid,ppid`
  - `sudo`
  - `sh`
  - `kill`
  - `cp`
  - `chmod`
  - `rm`
  - `ln`
  - `echo`
  - `exit`
  - `id`

- `uname`
- `grep`

- Your device may also require the following optional shell commands:
  - (Optional) `systemctl`. This command is used to set up the AWS IoT Greengrass Core software as a system service.
  - (Optional) `useradd`, `groupadd`, and `usermod`. These command are used to set up the `ggc_user` system user and `ggc_group` system group.
  - (Optional) `mkfifo`. This command is used to run Lambda functions as components.
- To configure system resource limits for component processes, your device must run Linux kernel version 2.6.24 or later.
- To run Lambda functions, your device must meet additional requirements. For more information, see Lambda function requirements.

> **Important**
>
> On Ubuntu 25.10 and later, the default `sudo` implementation does not support the `-E` (preserve environment) flag. Without this flag, the system does not forward environment variables to components that run as a different system user. As a result, those components fail to start.
> To resolve this issue, switch to the classic `sudo` implementation (`sudo.ws`). You must make this change because AWS IoT Greengrass uses the `-E` flag to pass environment variables to component processes.
>
> ```
> sudo update-alternatives --set sudo /usr/bin/sudo.ws
> ```

Windows

- The use of an AWS Region that supports AWS IoT Greengrass V2. For the list of supported Regions, see AWS IoT Greengrass V2 endpoints and quotas in the *AWS General Reference*.
- Minimum 256 MB disk space available for the AWS IoT Greengrass Core software. This requirement doesn't include components deployed to the core device.

- Minimum 160 MB RAM allocated to the AWS IoT Greengrass Core software. This requirement doesn't include components that run on the core device. For more information, see Control memory allocation with JVM options.
- Java Runtime Environment (JRE) version 8 or greater. Java must be available on the PATH system variable on the device. To use Java to develop custom components, you must install a Java Development Kit (JDK). We recommend that you use Amazon Corretto or OpenJDK long-term support versions. Version 8 or higher is required..

  > **Note**
  >
  > To use version 2.5.0 of the Greengrass nucleus, you must use a 64-bit version of the Java Runtime Environment (JRE). Greengrass nucleus version 2.5.1 supports 32-bit and 64-bit JREs.

- The user who installs the AWS IoT Greengrass Core software must be an administrator.
- You must install the AWS IoT Greengrass Core software as a system service. Specify `--setup-system-service true` when you install the software.
- Each user that runs component processes must exist in the LocalSystem account, and the user's name and password must be in the Credential Manager instance for the LocalSystem account. You can set up this user when you follow instructions to install the AWS IoT Greengrass Core software.
- The core device must be able to perform outbound requests to a set of endpoints and ports. For more information, see Allow device traffic through a proxy or firewall.

Supported platforms

AWS IoT Greengrass officially supports devices running the following platforms. Devices with platforms not included in this list might work, but AWS IoT Greengrass tests on only these specified platforms.

Linux

Architectures:

- Armv7l
- Armv8 (AArch64)
- x86_64

Windows

Architectures:

- x86_64

Versions:

- Windows 10
- Windows 11
- Windows Server 2019
- Windows Server 2022
- Windows Server 2025

> **Note**
>
> Some AWS IoT Greengrass features aren't currently supported on Windows devices. For more information, see Greengrass feature compatibility and Feature considerations.

Feature considerations

Some AWS IoT Greengrass features aren't currently supported on Windows devices. Review the feature differences to confirm if a Windows device satisfies your requirements. For more information, see Greengrass feature compatibility.

To build a custom Linux-based operating system, you can use the BitBake recipe for AWS IoT Greengrass in the `meta-aws` project. The `meta-aws` project provides recipes that you can use to build AWS edge software capabilities in embedded Linux systems that are built with OpenEmbedded and Yocto Project build frameworks. The Yocto Project is an open source collaboration project that helps you build custom Linux-based systems for embedded applications regardless of hardware architecture. The BitBake recipe for AWS IoT Greengrass installs, configures, and automatically runs the AWS IoT Greengrass Core software on your device.

Linux platforms can also run AWS IoT Greengrass in a Docker container. For more information, see Run AWS IoT Greengrass Core software in a Docker container.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

For more information, see Supported platforms.

Requirements

Devices must meet certain requirements to install and run the Greengrass nucleus and the AWS IoT Greengrass Core software. For more information, see Device requirements.

The Greengrass nucleus component is supported to run in a VPC. To deploy this component in a VPC, the following is required.

- The Greengrass nucleus component must have connectivity to AWS IoT data, AWS IoT Credentials, and Amazon S3.

Dependencies

The Greengrass nucleus does not include any component dependencies. However, several AWS-provided components include the nucleus as a dependency. For more information, see AWS-provided components.

For more information about component dependencies, see the component recipe reference.

Download and installation

You can download an installer that sets up the Greengrass nucleus component on your device. This installer sets up your device as a Greengrass core device. There are two types of installations that you can perform: a quick installation that creates required AWS resources for you, or a manual installation where you create the AWS resources yourself. For more information, see Install the AWS IoT Greengrass Core software.

You can also follow a tutorial to install the Greengrass nucleus and explore Greengrass component development. For more information, see Tutorial: Getting started with AWS IoT Greengrass V2.

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component. Some parameters require that the AWS IoT Greengrass Core software restarts to take effect. For more information about why and how to configure this component, see Configure the AWS IoT Greengrass Core software.

`iotRoleAlias`

The AWS IoT role alias that points to a token exchange IAM role. The AWS IoT credentials provider assumes this role to allow the Greengrass core device to interact with AWS services. For more information, see Authorize core devices to interact with AWS services.

When you run the AWS IoT Greengrass Core software with the `--provision true` option, the software provisions a role alias and sets its value in the nucleus component.

`interpolateComponentConfiguration`

(Optional) You can enable the Greengrass nucleus to interpolate component recipe variables in component configurations and merge configuration updates. We recommend that you set this option to `true` so that the core device can run Greengrass components that use recipe variables in their configurations.

This feature is available for v2.6.0 and later of this component.

Default: `false`

`networkProxy`

(Optional) The network proxy to use for all connections. For more information, see Connect on port 443 or through a network proxy.

> **Important**
>
> When you deploy a change to this configuration parameter, the AWS IoT Greengrass Core software restarts for the change to take effect.

This object contains the following information:

`noProxyAddresses`

(Optional) A comma-separated list of IP addresses or hostnames that are exempt from the proxy.

`proxy`

The proxy to which to connect. This object contains the following information:

`url`

The URL of the proxy server in the format `scheme://userinfo@host:port`.

- `scheme` – The scheme, which must be `http` or `https`.

  > **Important**
  >
  > Greengrass core devices must run Greengrass nucleus v2.5.0 or later to use HTTPS proxies.
  > If you configure an HTTPS proxy, you must add the proxy server CA certificate to the core device's Amazon root CA certificate. For more information, see Enable the core device to trust an HTTPS proxy.

- `userinfo` – (Optional) The user name and password information. If you specify this information in the `url`, the Greengrass core device ignores the `username` and `password` fields.
- `host` – The host name or IP address of the proxy server.
- `port` – (Optional) The port number. If you don't specify the port, then the Greengrass core device uses the following default values:
  - `http` – 80
  - `https` – 443

`username`

(Optional) The user name that authenticates the proxy server.

`password`

(Optional) The password that authenticates the proxy server.

`mqtt`

(Optional) The MQTT configuration for the Greengrass core device. For more information, see Connect on port 443 or through a network proxy.

> **Important**
>
> When you deploy a change to this configuration parameter, the AWS IoT Greengrass Core software restarts for the change to take effect.

This object contains the following information:

`port`

(Optional) The port to use for MQTT connections.

Default: `8883`

`keepAliveTimeoutMs`

(Optional) The amount of time in milliseconds between each `PING` message that the client sends to keep the MQTT connection alive. This value must be greater than `pingTimeoutMs`.

Default: `60000` (60 seconds)

`pingTimeoutMs`

(Optional) The amount of time in milliseconds that the client waits to receive a `PINGACK` message from the server. If the wait exceeds the timeout, the core device closes and reopens the MQTT connection. This value must be less than `keepAliveTimeoutMs`.

Default: `30000` (30 seconds)

> **Note**
>
> The effective maximum of this value depends on the MQTT version that you use. When you use `mqtt3`, the core device limits this value to `32767` (approximately 32.7 seconds) and silently reduces larger values to this limit. When you use `mqtt5`, the core device caps the effective ping timeout at half of the keep-alive interval that the server negotiates. The core device uses the value that you set only if it is less than `keepAliveTimeoutMs` divided by 2.

`operationTimeoutMs`

(Optional) The amount of time in milliseconds that the client waits for MQTT operations (such as `CONNECT` or `PUBLISH`) to complete. This option doesn't apply to MQTT `PING` or keep alive messages.

Default: `30000` (30 seconds)

`maxInFlightPublishes`

(Optional) The maximum number of unacknowledged MQTT QoS 1 messages that can be in flight at the same time.

This feature is available for v2.1.0 and later of this component.

Default: 5

Valid range: Maximum value of 100

`maxMessageSizeInBytes`

(Optional) The maximum size of an MQTT message. If a message exceeds this size, the Greengrass nucleus rejects the message with an error.

This feature is available for v2.1.0 and later of this component.

Default: `131072` (128 KB)

Valid range: Maximum value of `2621440` (2.5 MB)

`maxPublishRetry`

(Optional) The maximum number of times to retry a message that fails to publish. You can specify `-1` to retry unlimited times.

This feature is available for v2.1.0 and later of this component.

Default: `100`

`standaloneMqttTimeoutMs`

(Optional) The time, in milliseconds, that the nucleus waits for standalone MQTT connections during IoT connectivity switch operations. This timeout covers the pre-flight connectivity check and the status report that the nucleus sends to the source account after a successful switch.

This feature requires v2.18.0 or later of this component.

Default: `60000` (60 seconds)

Valid range: Minimum value of 1

`spooler`

(Optional) The MQTT spooler configuration for the Greengrass core device. This object contains the following information:

`storageType`

The storage type for storing messages. If `storageType` is set to `Disk`, the `pluginName` can be configured. You can specify either `Memory` or `Disk`.

This feature is available for v2.11.0 and later of the Greengrass nucleus component.

> **Important**
>
> If the MQTT spooler `storageType` is set to `Disk` and you want to downgrade Greengrass nucleus from version 2.11.x to an earlier version, you must change the configuration back to `Memory`. The only configuration for `storageType` that is supported in Greengrass nucleus versions 2.10.x and earlier is `Memory`. Not following this guidance can result in the spooler breaking. This would cause your Greengrass core device to not be able to send MQTT messages to the AWS Cloud.

Default: `Memory`

`pluginName`

(Optional) The plugin component name. This component will only be used if `storageType` is set to `Disk`. This option defaults to `aws.greengrass.DiskSpooler` and will use the Greengrass-provided Disk spooler.

This feature is available for v2.11.0 and later of the Greengrass nucleus component.

Default: `"aws.greengrass.DiskSpooler"`

`maxSizeInBytes`

(Optional) The maximum size of the cache where the core device stores unprocessed MQTT messages in memory. If the cache is full, new messages are rejected.

Default: `2621440` (2.5 MB)

`keepQos0WhenOffline`

(Optional) You can spool MQTT QoS 0 messages that the core device receives while its offline. If you set this option to `true`, the core device spools QoS 0 messages that it can't send while it's offline. If you set this option to `false`, the core device discards these messages. The core device always spools QoS 1 messages unless the spool is full.

Default: `false`

`version`

(Optional) The version of MQTT. You can specify either `mqtt3` or `mqtt5`.

This feature is available for v2.10.0 and later of the Greengrass nucleus component.

Default: `mqtt5`

`receiveMaximum`

(Optional) The maximum number of unacknowledged QoS1 packets the broker can send.

This feature is available for v2.10.0 and later of the Greengrass nucleus component.

Default: `100`

`sessionExpirySeconds`

(Optional) The amount of time in seconds you can request for a session to last from IoT Core. The default is the maximum time supported by AWS IoT Core.

This feature is available for v2.10.0 and later of the Greengrass nucleus component.

Default: `604800 (7 days)`

`minimumReconnectDelaySeconds`

(Optional) An option for reconnection behavior. The minimum amount of time in seconds for MQTT to reconnect.

This feature is available for v2.10.0 and later of the Greengrass nucleus component.

Default: `1`

`maximumReconnectDelaySeconds`

(Optional) An option for reconnection behavior. The maximum amount of time in seconds for MQTT to reconnect.

This feature is available for v2.10.0 and later of the Greengrass nucleus component.

Default: `120`

> **Note**
>
> The core device applies this option only when you set the MQTT version to `mqtt5`. When you use `mqtt3`, the core device ignores this value and uses a fixed maximum reconnect delay of 128 seconds.

`minimumConnectedTimeBeforeRetryResetSeconds`

(Optional) An option for reconnection behavior. The amount of time in seconds a connection must be active before the retry delay is reset back to the minimum.

This feature is available for v2.10.0 and later of the Greengrass nucleus component.

Default: `30`

`jvmOptions`

(Optional) The JVM options to use to run the AWS IoT Greengrass Core software. For information about recommended JVM options for running AWS IoT Greengrass Core software, see Control memory allocation with JVM options.

> **Important**
>
> When you deploy a change to this configuration parameter, the AWS IoT Greengrass Core software restarts for the change to take effect.

`iotDataEndpoint`

The AWS IoT data endpoint for your AWS account.

When you run the AWS IoT Greengrass Core software with the `--provision true` option, the software gets your data and credentials endpoints from AWS IoT and sets them in the nucleus component.

`iotCredEndpoint`

The AWS IoT credentials endpoint for your AWS account.

When you run the AWS IoT Greengrass Core software with the `--provision true` option, the software gets your data and credentials endpoints from AWS IoT and sets them in the nucleus component.

`greengrassDataPlaneEndpoint`

This feature is available in v2.7.0 and later of this component.

(Optional) The Greengrass data plane endpoint to use. Set to **`iotdata`** to use the same endpoint as the IoT data endpoint specified in `iotDataEndpoint`.

You must set this option to **`iotdata`** in the following cases:

- You use a device certificate signed by a private certificate authority (CA). For more information, see Use a device certificate signed by a private CA.
- You deploy nucleus configuration updates to switch the device's IoT connectivity to a different AWS account or Region. Without this setting, the data plane remains bound to the original account after the connectivity switch. This breaks component artifact downloads and control plane operations. For more information, see Switch a Greengrass core device to a different AWS account or Region.

You can set this option during deployment or installation using the **`--init-config`** installer argument.

Default: `""` (empty string – uses the Greengrass service endpoint)

`greengrassDataPlanePort`

This feature is available in v2.0.4 and later of this component.

(Optional) The port to use for data plane connections. For more information, see Connect on port 443 or through a network proxy.

> **Important**
>
> You must specify a port where the device can make outbound connections. If you specify a port that is blocked, the device won't be able to connect to AWS IoT Greengrass to receive deployments.

Choose from the following options:

- 443
- 8443

Default: 8443

`awsRegion`

The AWS Region to use.

`runWithDefault`

The system user to use to run components.

> **Important**
>
> When you deploy a change to this configuration parameter, the AWS IoT Greengrass Core software restarts for the change to take effect.

This object contains the following information:

`posixUser`

The name or ID of the system user and, optionally, system group that the core device uses to run generic and Lambda components. Specify the user and group separated by a colon (`:`) in the following format: `user:group`. The group is optional. If you don't specify a group, the AWS IoT Greengrass Core software uses the primary group for the user. For example, you can specify `ggc_user` or `ggc_user:ggc_group`. For more information, see Configure the user that runs components.

When you run the AWS IoT Greengrass Core software installer with the `--component-default-user` *`ggc_user:ggc_group`* option, the software sets this parameter in the nucleus component.

`windowsUser`

This feature is available in v2.5.0 and later of this component.

The name of the Windows user to use to run this component on Windows core devices. The user must exist on each Windows core device, and its name and password must be stored in the LocalSystem account's Credentials Manager instance. For more information, see Configure the user that runs components.

When you run the AWS IoT Greengrass Core software installer with the `--component-default-user` *`ggc_user`* option, the software sets this parameter in the nucleus component.

`systemResourceLimits`

This feature is available in v2.4.0 and later of this component. AWS IoT Greengrass doesn't currently support this feature on Windows core devices.

The system resource limits to apply to generic and non-containerized Lambda component processes by default. You can override system resource limits for individual components when you create a deployment. For more information, see Configure system resource limits for components.

This object contains the following information:

`cpus`

The maximum amount of CPU time that each component's processes can use on the core device. A core device's total CPU time is equivalent to the device's number of CPU cores. For example, on a core device with 4 CPU cores, you can set this value to 2 to limit each component's processes to 50 percent usage of each CPU core. On a device with 1 CPU core, you can set this value to `0.25` to limit each component's processes to 25 percent usage of the CPU. If you set this value to a number greater than the number of CPU cores, the AWS IoT Greengrass Core software doesn't limit the components' CPU usage.

`memory`

The maximum amount of RAM (in kilobytes) that each component's processes can use on the core device.

`s3EndpointType`

(Optional) The S3 endpoint type. This parameter will only take effect for the US East (N. Virginia) (`us-east-1`) Region. Setting this parameter from any other Region will be ignored. Choose from the following options:

- `REGIONAL` – S3 client and presigned URL uses the regional endpoint.
- `GLOBAL` – S3 client and presigned URL uses the legacy endpoint.
- `DUALSTACK` – S3 presigned URL uses the dualstack endpoint.

Default: `GLOBAL`

`fipsMode`

(Optional) Causes Greengrass to use FIPS endpoints. For more information on how to enable FIPS endpoints, see FIPS endpoints.

Choose from the following options:

- `true` When set to true the endpoints will use FIPS endpoint.
- `false` When false the endpoints will not use FIPS endpoint.

Default: `false`

`logging`

(Optional) The logging configuration for the core device. For more information about how to configure and use Greengrass logs, see Monitor AWS IoT Greengrass logs.

This object contains the following information:

`level`

(Optional) The minimum level of log messages to output.

Choose from the following log levels, listed here in level order:

- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`

Default: `INFO`

`format`

(Optional) The data format of the logs. Choose from the following options:

- `TEXT` – Choose this option if you want to view logs in text form.
- `JSON` – Choose this option if you want to view logs with the Greengrass CLI logs command or interact with logs programmatically.

Default: `TEXT`

`outputType`

(Optional) The output type for logs. Choose from the following options:

- `FILE` – The AWS IoT Greengrass Core software outputs logs to files in the directory that you specify in `outputDirectory`.
- `CONSOLE` – The AWS IoT Greengrass Core software prints logs to `stdout`. Choose this option to view logs as the core device prints them.

Default: `FILE`

`fileSizeKB`

(Optional) The maximum size of each log file (in kilobytes). After a log file exceeds this maximum file size, the AWS IoT Greengrass Core software creates a new log file.

This parameter applies only when you specify `FILE` for `outputType`.

Default: `1024`

`totalLogsSizeKB`

(Optional) The maximum total size of log files (in kilobytes) for each component, including the Greengrass nucleus. The Greengrass nucleus' log files also include logs from plugin components. After a component's total size of log files exceeds this maximum size, the AWS IoT Greengrass Core software deletes that component's oldest log files.

This parameter is equivalent to the log manager component's disk space limit parameter (`diskSpaceLimit`), which you can specify for the Greengrass nucleus (system) and each component. The AWS IoT Greengrass Core software uses the minimum of the two values as the maximum total log size for the Greengrass nucleus and each component.

This parameter applies only when you specify `FILE` for `outputType`.

Default: `10240`

`outputDirectory`

(Optional) The output directory for log files.

This parameter applies only when you specify `FILE` for `outputType`.

Default: *`/greengrass/v2`*`/logs`, where *`/greengrass/v2`* is the AWS IoT Greengrass root folder.

`fleetstatus`

This parameter is available in v2.1.0 and later of this component.

(Optional) The fleet status configuration for the core device.

This object contains the following information:

`periodicStatusPublishIntervalSeconds`

(Optional) The amount of time (in seconds) between which the core device publishes device status to the AWS Cloud.

Minimum: `86400` (24 hours)

Default: `86400` (24 hours)

`telemetry`

(Optional) The system health telemetry configuration for the core device. For more information about telemetry metrics and how to act on telemetry data, see Gather system health telemetry data from AWS IoT Greengrass core devices.

This object contains the following information:

`enabled`

(Optional) You can enable or disable telemetry.

Default: `true`

`periodicAggregateMetricsIntervalSeconds`

(Optional) The interval (in seconds) over which the core device aggregates metrics.

If you set this value lower than the minimum supported value, the nucleus uses the default value instead.

Minimum: `3600`

Default: `3600`

`periodicPublishMetricsIntervalSeconds`

(Optional) The amount of time (in seconds) between which the core device publishes telemetry metrics to the AWS Cloud.

If you set this value lower than the minimum supported value, the nucleus uses the default value instead.

Minimum: `86400`

Default: `86400`

`deploymentPollingFrequencySeconds`

(Optional) The period in seconds at which to poll for deployment notifications.

Default: `15`

`componentStoreMaxSizeBytes`

(Optional) The maximum size on disk of the component store, which comprises component recipes and artifacts.

Default: `10000000000` (10 GB)

`bootConfigTlogCompactionThresholdBytes`

This parameter is available in v2.18.3 and later of this component.

(Optional) The size threshold, in bytes, at which the Greengrass nucleus compacts its configuration transaction log (`config.tlog`) when the core device starts.

Compaction does not change your configuration. It removes only outdated entries from the log, which reduces startup time and disk usage on the core device.

To disable boot-time compaction, set this value to `0` or a negative value.

Specify the threshold as a number of bytes, such as `10000000`. The nucleus also accepts scientific notation, such as `1e7`. If you set a value that isn't a number, such as `20MB`, the nucleus uses the default value and records a warning in its logs. An invalid value doesn't turn off compaction.

> **Note**
>
> Compaction does not prevent your device from starting. If the nucleus can't compact the log, the core device starts normally with the uncompacted log and records the error in its logs. The nucleus tries again the next time the core device starts.

Default: `10000000` (10 MB)

`platformOverride`

(Optional) A dictionary of attributes that identify the core device's platform. Use this to define custom platform attributes that component recipes can use to identify the correct lifecycle and

artifacts for the component. For example, you might define a hardware capability attribute to deploy only the minimal set of artifacts for a component to run. For more information, see the manifest platform parameter in the component recipe.

You can also use this parameter to override the `os` and `architecture` platform attributes of the core device.

`httpClient`

This parameter is available in v2.5.0 and later of this component.

(Optional) The HTTP client configuration for the core device. These configuration options apply to all HTTP requests made by this component. If a core device runs on a slower network, you can increase these timeout durations to prevent HTTP requests from timing out.

This object contains the following information:

`connectionTimeoutMs`

(Optional) The amount of time (in milliseconds) to wait for a connection to open before the connection request times out.

Default: `2000` (2 seconds)

`socketTimeoutMs`

(Optional) The amount of time (in milliseconds) to wait for data to transfer over an open connection before the connection times out.

Default: `30000` (30 seconds)

`deploymentConfigurationTimeSource`

This parameter is available in v2.15.0 and later of this component.

(Optional) The timestamp to use when processing a deployment. The default is the `deploymentCreationTime`.

This object contains the following values:

`deploymentCreationTime`

The default value of `deploymentConfigurationTimeSource`. The device uses the deployment creation timestamp to resolve configuration key conflicts during processing.

When this behavior is selected, local device configuration held by the nucleus may have a greater timestamp than that of the incoming deployment and rejects incoming configuration changes which are now considered outdated.

`deploymentProcessingTime`

The device uses its local timestamp to resolve configuration key conflicts during deployment processing. When processed, the device updates configurations based on the processing timestamp rather than the deployment creation timestamp. This behavior assumes the device clock is properly calibrated.

Configure this nucleus setting in your initial device image or installation rather than through a deployment when you want new devices to use this behavior on first connection. Use the `--init-config` option of the nucleus classic installer for this configuration.

This initial configuration is essential because devices process multiple deployments in arbitrary order. Without proper initial configuration, a device might process deployments using the default `deploymentCreationTime` behavior before receiving the deployment that sets the nucleus configuration to `deploymentProcessingTime`.

Example: Configuration merge update

```
{
  "iotRoleAlias": "GreengrassCoreTokenExchangeRoleAlias",
  "networkProxy": {
    "noProxyAddresses": "http://192.168.0.1,www.example.com",
    "proxy": {
      "url": "http://my-proxy-server:1100",
      "username": "Mary_Major",
      "password": "pass@word1357"
    }
  },
  "mqtt": {
    "port": 443
  },
  "greengrassDataPlanePort": 443,
  "jvmOptions": "-Xmx64m",
  "runWithDefault": {
    "posixUser": "ggc_user:ggc_group"
  }
}
```

Local log file

This component uses the following log file.

Linux

```
/greengrass/v2/logs/greengrass.log
```

Windows

```
C:\greengrass\v2\logs\greengrass.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
  ```

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.18.4	Bug fixes and improvements • Fixes an issue where the nucleus fails a deployment instead of retrying when a Docker image download is interrupted by a network error or fails for an unrecognized reason.

Version	Changes
2.18.3	New features • Adds the `bootConfigTlogCompactionThresholdBytes` configuration parameter to enable configuration transaction log compaction at boot, which reclaims disk space on core devices. Bug fixes and improvements • Fixes an issue where a failed first deployment could cause the nucleus to lose device group membership and remove installed components after a restart.
2.18.2	Bug fixes and improvements • Fixes an issue where the nucleus could write unchanged configuration values to the configuration store.
2.18.1	Bug fixes and improvements • Fixes an issue where the nucleus might fail to fetch credentials after a transient connection failure.
2.18.0	New features • Adds support for switching a core device's AWS IoT connectivity to a different AWS account or region through a nucleus configuration deployment. The nucleus validates connectivity to the new endpoints before it applies the change, and reports the terminal deployment status to the source account. • Adds support for Windows Server 2025. Bug fixes and improvements • Fixes a startup timing issue on systemd devices by starting the AWS IoT Greengrass Core software only after the system clock synchronizes. • Fixes an issue where a component might encounter stale subscript ions after it closes a local IPC stream during an in-progress subscribe request. • General bug fixes and improvements.

Version	Changes
2.17.0	New features • Installs all Amazon Root CA certificates during auto-provisioning. • Adds an uninstall lifecycle that runs when a deployment removes a component. • Allows installing Greengrass Core software as a regular Linux user.
2.16.1	Bug fixes and improvements • Adds configuration for credential retry intervals after Token Exchange Service failures.
2.16.0	New features • Nucleus now supports cgroups V2. Bug fixes and improvements • Fixes an issue where the nucleus would not clean up stale deployments when a deployment was cancelled.
2.15.1	Bug fixes and improvements • Fixes an issue where the nucleus failed to clean up component processes after the 30-second shutdown timeout during bootstrap deployments.

Version	Changes
2.15.0	New features • Adds telemetry feature to include host system information like CPU and OS details. For more information, see telemetry component page. • Adds `deploymentConfigurationTimeSource` configuration. For more information, see nucleus component configuration. Bug fixes and improvements • Nucleus now prioritizes the use of local versions and artifacts by default, improving deployment consistency and reducing external dependencies. Users can override this behavior by specifying alternati ve preferences in the deployment document. • Adds a new warning log when a provided Windows user doesn't match the character set normally accepted by Windows.
2.14.3	Bug fixes and improvements • Allows the token exchange service to restart on port configuration changes. • Fixes an issue where fleet status service doesn't send component status change messages for unpinned lambdas if they aren't triggered. • Fixes an issue where components aren't gracefully shutdown when a new component version is deployed. • Fixes an issue where built-in plugins' injection hooks are executed twice resulting in extra lifecycle events and duplicate logging. • Improves component lifecycle logging on Windows devices.
2.14.2	Bug fixes and improvements • Fixes an issue where an HTTP client isn't configured with mutual auth.
2.14.1	Bug fixes and improvements • Fixes an issue where components weren't stopped properly on new installations of Greengrass.

Version	Changes
2.14.0	**Warning** This version is no longer available. The improvements in this version are available in later versions of this component. New features • New dual-stack endpoint support enables IPv6 network communica tion. • Enhanced resilience against nucleus restart failures and Launchdirectory corruption. Bug fixes and improvements • Fixes memory leaks in IPC PubSub subscription closures. • Fixes run lifecycle of the component where it enters into ERRORED state due to startup timeout when skipif condition is true. • Fixes an issue where the core device fails to connect to AWS IoT Core when the TLS policy is set to TLS13_1_3_2022_10.
2.13.0	New features • Support FIPS endpoint in Nucleus. Bug fixes and improvements • Cancel deployment improvements: Deployments can now be cancelled while a new configuration is being merged and while waiting for services to start.
2.12.6	Bug fixes and improvements • Fixes an issue that causes a crash at startup on certain ARMv8 processors, including the Jetson Nano.

Version	Changes
2.12.5	Bug fixes and improvements • Fixes an issue where deployment rollback occasionally gets stuck while rolling back a previously broken component with hard dependencies. • Fixes an issue where the nucleus doesn't publish status updates after fleet provisioning. • Adds retries for the `GetDeploymentConfiguration` API after getting 404 errors.
2.12.4	Bug fixes and improvements • Fixes an issue where the nucleus enters a deadlock condition during startup on some Linux devices.
2.12.3	**Warning** This version is no longer available. The improvements in this version are available in later versions of this component. Bug fixes and improvements • Fixes an issue where the nucleus doesn't report the correct component status after the nucleus relaunches and during component recovery. • General bug fixes and improvements.
2.12.2	Bug fixes and improvements • Fixes an issue where old logs weren't cleaned up properly. • General bug fixes and improvements.
2.12.1	Bug fixes and improvements • Fixes an issue where the nucleus may duplicate MQTT subscriptions to deployment topics leading to additional logging and MQTT publishes.

Version	Changes
2.12.0	New features • Enables you to run the bootstrap lifecycle steps as part of a rollback deployment.
2.11.3	Bug fixes and improvements • Fixes an issue in the nucleus where it may improperly start a component when its dependencies fail. New features • Adds configurable s3 endpoint type.
2.11.2	Bug fixes and improvements • Fixes an issue in the nucleus MQTT 5 client where it may appear offline when a large number (> 50) of subscriptions are in use. • Adds a retry for the docker dial TCP failure.
2.11.1	Bug fixes and improvements • Fixes an issue where the nucleus doesn't start if a bootstrap task fails and the deployment metadata file is corrupted. • Fixes an issue where on-demand Lambda components aren't reported in deployment status updates. • Adds support for duplicate authorization policy IDs.
2.11.0	New features • Enables you to cancel a local deployment. • Enables you to configure a failure handling policy for a local deployment. • Adds support for a disk spooler plugin.
2.10.3	Bug fixes and improvements • Fixes an issue where Greengrass doesn't subscribe to deployment notifications when using the PKCS#11 provider.

Version	Changes
2.10.2	Bug fixes and improvements • Allows case insensitive parsing of component lifecycles. • Fixes an issue where the environment PATH variable was not recreated correctly. • Fixes proxy URI encoding for components including stream manager for usernames with special characters.
2.10.1	Bug fixes and improvements • Fixes an issue that could cause a crash at startup on certain ARMv8 processors, including the Jetson Nano. • Greengrass no longer closes a component's standard in, this reverts the behavior to the pre-2.10.0 behavior

Version	Changes
2.10.0	New features • Adds `interpolateComponentConfiguration` support for the empty regular expression. Greengrass now interpolates from the root config object. • Adds support for MQTT5. • Adds a mechanism for loading plugin components quickly without scanning. • Enables Greengrass to save disk space by deleting unused Docker images. Bug fixes and improvements • Fixes an issue where rollback leaves certain configuration values in place from a deployment. • Fixes an issue where the Greengrass nucleus validates for an AWS domain sequence in custom non-AWS credentials and data endpoints. • Updates multi-group dependency resolution to re-resolve all group dependencies via AWS Cloud negotiation, instead of locking to the active version. This update also removes the deployment error code `INSTALLED_COMPONENT_NOT_FOUND` . • Updates the Greengrass nucleus to skip downloading Docker images when they already exist locally. • Updates the Greengrass nucleus to restart a component install step before timeout expires. • Additional minor fixes and improvements.
2.9.6	Bug fixes and improvements • Fixes an issue where a Greengrass deployment fails with the error LAUNCH_DIRECTORY_CORRUPTED and a subsequent device reboot fails to start Greengrass. This error may occur when you move the Greengrass device between multiple thing groups with deployments that require Greengrass to restart.

Version	Changes
2.9.5	New features • Adds support for Greengrass nucleus software signature verification. Bug fixes and improvements • Fixes an issue where a deployment fails when the local recipe metadata region doesn't match the Greengrass nucleus launch region. The Greengrass nucleus now renegotiates with the cloud when this happens. • Fixes an issue where the MQTT message spooler fills up and never removes messages. • Additional minor fixes and improvements.
2.9.4	Bug fixes and improvements • Checks for a null message before it drops QOS 0 messages. • Truncates job status detail values if they exceed the 1024 character limit. • Updates the bootstrap script for Windows to correctly read the Greengrass root path if that path includes spaces. • Updates subscribing to AWS IoT Core so that it drops client messages if the subscription response wasn't sent. • Ensures that the nucleus loads its configuration from backup files when the main configuration file is corrupt or missing.
2.9.3	Bug fixes and improvements • Ensures MQTT client IDs aren't duplicated. • Adds more robust file-reading and writing to avoid and recover from corruption. • Retries docker image pull on specific network-related errors. • Adds the `noProxyAddresses` option for MQTT connection.

Version	Changes
2.9.2	Bug fixes and improvements • Fixes an issue where configuring `interpolateComponentConfiguration` doesn't apply to an ongoing deployment. • Uses OSHI to list all child processes.
2.9.1	Bug fixes and improvements • Adds fix where Greengrass restarts if a deployment removes a plugin component.
2.9.0	New features • Adds the ability to create subdeployments that retry deployments with a smaller subset of devices. This feature creates a more efficient way to test and resolve unsuccessful deployments. Bug fixes and improvements • Improves support for systems that don't have `useradd`, `groupadd`, and `usermod`. • Additional minor fixes and improvements.
2.8.1	Bug fixes and improvements • Fixes an issue where deployment error codes were not generated correctly from Greengrass API errors. • Fixes an issue where fleet status updates send inaccurate information when a component reaches an `ERRORED` state during a deployment. • Fixes an issue where deployments couldn’t complete when Greengrass had more than 50 existing subscriptions.

Version	Changes
2.8.0	New features • Updates the Greengrass nucleus to report a deployment health status response that includes detailed error codes when there is a problem deploying components to a core device. For more information, see Detailed deployment error codes. • Updates the Greengrass nucleus to report a component health status response that includes detailed error codes when a component enters the BROKEN or ERRORED state. For more information, see Detailed component status codes. • Expands status message fields to improve cloud availability informati on for devices. • Improves fleet status service robustness. Bug fixes and improvements • Allows a broken component to reinstall when its configuration changes. • Fixes an issue where a nucleus restart during bootstrap deployment causes a deployment to fail. • Fixes an issue in Windows where installation fails when a root path contains spaces. • Fixes an issue where a component shut down during a deployment uses the shutdown script of the new version. • Various shutdown improvements. • Additional minor fixes and improvements.

Version	Changes
2.7.0	New features • Updates the Greengrass nucleus to send status updates to the AWS IoT Greengrass cloud when the core device applies a local deployment. • Adds support for client certificates signed by a custom certificate authority (CA), where the CA isn't registered with AWS IoT. To use this feature, you can set the new `greengrassDataPlaneEndpoint` configuration option to `iotdata`. For more information, see Use a device certificate signed by a private CA. Bug fixes and improvements • Fixes an issue where the Greengrass nucleus rolls back a deployment in certain scenarios when the nucleus is stopped or restarted. The nucleus now resumes the deployment after the nucleus restarts. • Updates the Greengrass installer to respect the `--start` argument when you specify to set up the software as a system service. • Updates the behavior of SubscribeToComponentUpdates to set the deployment ID in events where the nucleus updated a component. • Additional minor fixes and improvements.

Version	Changes
2.6.0	New features • Adds support for MQTT wildcards when you subscribe to local publish/subscribe topics. For more information, see Publish/subscribe local messages and SubscribeToTopic. • Adds support for recipe variables in component configurations, other than the *component_dependency_name* `:configuration:` *json_pointer* recipe variable. You can use these recipes variables when you define a component's `DefaultConfiguration` in a recipe or when you configure a component in a deployment. To enable this feature, set the interpolateComponentConfiguration configuration option to `true`. For more information, see Recipe variables and Use recipe variables in merge updates. • Adds full support for the `*` wildcard in interprocess communication (IPC) authorization policies. You can now specify the `*` character in a resource string to match any combination of characters. For more information, see Wildcards in authorization policies. • Adds support for custom components to call IPC operations that the Greengrass CLI uses. You can use these IPC operations to manage local deployments, view component details, and generate a password that you can use to sign in to the local debug console. For more information, see IPC: Manage local deployments and components. Bug fixes and improvements • Fixes an issue where dependent components wouldn't react when their hard dependencies restart or change states in certain scenarios. • Improves error messages that the core device reports to the AWS IoT Greengrass cloud service when a deployment fails. • Fixes an issue where the Greengrass nucleus applied a thing deployment twice in certain scenarios when the nucleus restarts. • Additional minor fixes and improvements. For more information, see the releases on GitHub.

Version	Changes
2.5.6	New features • Adds support for hardware security modules that use ECC keys. You can use a hardware security module (HSM) to securely store the device's private key and certificate. For more information, see Hardware security integration. Bug fixes and improvements • Fixes an issue where the deployment never completes when you deploy a component with a broken install script in certain scenarios. • Improves performance during startup. • Additional minor fixes and improvements.
2.5.5	New features • Adds the `GG_ROOT_CA_PATH` environment variable for component s, so you can access the root certificate authority (CA) certificate in custom components. Bug fixes and improvements • Adds support for Windows devices that use a display language other than English. • Updates how the Greengrass nucleus parses Boolean installer arguments, so you can specify a Boolean argument without a Boolean value to specify a `true` value. For example, you can now specify `--provision` instead of `--provision true` to install with automatic resource provisioning. • Fixes an issue where the core device didn't report its status to the AWS IoT Greengrass cloud service after provisioning in certain scenarios. • Additional minor fixes and improvements.
2.5.4	Bug fixes and improvements • General bug fixes and improvements.

Version	Changes
2.5.3	New features • Adds support for hardware security integration. You can use a hardware security module (HSM) to securely store the device's private key and certificate. For more information, see Hardware security integration. Bug fixes and improvements • Fixes an issue with runtime exceptions while the nucleus establishes MQTT connections with AWS IoT Core.
2.5.2	Bug fixes and improvements • Fixes an issue where after the Greengrass nucleus updates, the Windows service fails to start again after you stop it or reboot the device.
2.5.1	**Warning** This version is no longer available. The improvements in this version are available in later versions of this component. Bug fixes and improvements • Adds support for 32-bit versions of the Java Runtime Environment (JRE) on Windows. • Changes thing group removal behavior for core devices whose AWS IoT policy doesn't grant the `greengrass:ListThingGroupsForCoreDevice` permission. With this version, the deployment continues, logs a warning, and doesn't remove components when you remove the core device from a thing group. For more information, see Deploy AWS IoT Greengrass components to devices. • Fixes an issue with system environment variables that the Greengrass nucleus makes available to Greengrass component processes. You can now restart a component for it to use the latest system environment variables.

Version	Changes
2.5.0	New features • Adds support for core devices that run Windows. • Change the behavior of thing group removal. With this version, you can remove a core device from a thing group to uninstall that thing group's components in the next deployment. As a result of this change, a core device's AWS IoT policy must have the `greengrass:ListThingGroupsForCoreDevice` permission. If you used the AWS IoT Greengrass Core software installer to provision resources, the default AWS IoT policy allows `greengrass:*` , which includes this permission. For more information, see Device authentication and authorization for AWS IoT Greengrass. • Adds support for HTTPS proxy configurations. For more information, see Connect on port 443 or through a network proxy. • Adds the new `windowsUser` configuration parameter. You can use this parameter to specify the default user to use to run components on a Windows core device. For more information, see Configure the user that runs components. • Adds the new `httpClient` configuration options that you can use to customize HTTP request timeouts to improve performance on slow networks. For more information, see the httpClient configuration parameter. Bug fixes and improvements • Fixes the bootstrap lifecycle option to restart the core device from a component. • Adds support for hyphens in recipe variables. • Fixes IPC authorization for on-demand Lambda function components. • Improves log messages and changes non-critical logs from `INFO` to `DEBUG` level, so logs are more useful. • Removes the `iot:DescribeCertificate` permission from the default token exchange role that the Greengrass nucleus creates when

Version	Changes
	you install the AWS IoT Greengrass Core software with automatic provisioning. This permission isn't used by the Greengrass nucleus. • Fixes an issue so that the automatic provisioning script doesn't require the `iam:GetPolicy` permission if `iam:CreatePolicy` is available for the same policy. • Additional minor fixes and improvements.
2.4.0	New features • Adds support for system resource limits. You can configure the maximum amount of CPU and RAM usage that each component 's processes can use on the core device. For more information, see Configure system resource limits for components. • Adds IPC operations to pause and resume components. For more information, see PauseComponent and ResumeComponent. • Adds support for provisioning plugins. You can specify a JAR file to run during installation to provision required AWS resources for a Greengrass core device. The Greengrass nucleus includes an interface that you can implement to develop custom provisioning plugins. For more information, see Install AWS IoT Greengrass Core software with custom resource provisioning. • Adds the optional `thing-name-policy` argument to the AWS IoT Greengrass Core software installer. You can use this option to specify an existing or custom AWS IoT policy when you install the AWS IoT Greengrass Core software with automatic resource provisioning. Bug fixes and improvements • Updates logging configuration on startup. This fixes an issue where the logging configuration wasn't applied on startup. • Updates the nucleus loader symlink to point to the component store in the Greengrass root folder during installation. This update enables you to delete the JAR file and other nucleus artifacts that you download when you install the AWS IoT Greengrass Core software. • Additional minor fixes and improvements. For more information, see the releases on GitHub.

Version	Changes
2.3.0	New features • Adds support for deployment configuration documents up to 10 MB, up from 7 KB (for deployments that target things) or 31 KB (for deployments that target thing groups). To use this feature, a core device's AWS IoT policy must allow the `greengrass:GetDeploymentConfiguration` permission. If you used the AWS IoT Greengrass Core software installer to provision resources, your core device's AWS IoT policy allows `greengrass:*`, which includes this permission. For more information, see Device authentication and authorization for AWS IoT Greengrass. • Adds the `iot:thingName` recipe variable. You can use this recipe variable to get the name of the core device's AWS IoT thing in a recipe. For more information, see Recipe variables. Bug fixes and improvements • Additional minor fixes and improvements. For more information, see the releases on GitHub.
2.2.0	New features • Adds IPC operations for local shadow management. Bug fixes and improvements • Reduces the size of the JAR file. • Reduces memory usage. • Fixes issues where the log configuration wasn't updated in certain cases. • Additional minor fixes and improvements. For more information, see the releases on GitHub.

Version	Changes

2.1.0	New features
	• Supports downloading Docker images from private repositories in Amazon ECR.
	• Adds the following parameters to customize the MQTT configuration on core devices:
	  • `maxInFlightPublishes` – The maximum number of unacknowledged MQTT QoS 1 messages that can be in flight at the same time.
	  • `maxPublishRetry` – The maximum number of times to retry a message that fails to publish.
	• Adds the `fleetstatusservice` configuration parameter to configure the interval at which the core device publishes device status to the AWS Cloud.
	• Additional minor fixes and improvements. For more information, see the releases on GitHub.
	Bug fixes and improvements
	• Fixes an issue that caused shadow deployments to be duplicated when the nucleus restarts.
	• Fixes an issue that caused the nucleus to crash when it encountered a service load exception.
	• Improves component dependency resolution to fail a deployment that includes a circular dependency.
	• Fixes an issue that prevented a plugin component from being redeployed if that component had been previously removed from the core device.
	• Fix an issue that caused the `HOME` environment variable to be set to the *`/greengrass/v2`* `/work` directory for Lambda components or for components that run as root. The `HOME` variable is now correctly set to the home directory for the user that runs the component.
	• Additional minor fixes and improvements. For more information, see the releases on GitHub.

Version	Changes
2.0.5	Bug fixes and improvements • Correctly routes traffic through a configured network proxy when downloading AWS-provided components. • Use the correct Greengrass data plane endpoint in AWS China Regions.
2.0.4	New features • Enables HTTPS traffic over port 443. You can use the new `greengrasDataPlanePort` configuration parameter for version 2.0.4 of the nucleus component to configure HTTPS communication to travel over port 443 instead of the default port 8443. For more information, see Configure HTTPS over port 443. • Adds the work path recipe variable. You can use this recipe variable to get the path to components' work folders, which you can use to share files between components and their dependencies. For more information, see the work path recipe variable. Bug fixes and improvements • Prevents the creation of the token exchange AWS Identity and Access Management (IAM) role policy if a role policy already exists. As a result of this change, the installer now requires the `iam:GetPolicy` and `sts:GetCallerIdentity` when run with `--provision true`. For more information, see Minimal IAM policy for installer to provision resources. • Correctly handles the cancellation of a deployment that has not yet been registered successfully. • Updates the configuration to remove older entries with newer timestamps when rolling back a deployment. • Additional minor fixes and improvements. For more information, see the releases on GitHub.
2.0.3	Initial version.

Greengrass nucleus lite

The Greengrass nucleus lite (`aws.greengrass.NucleusLite`) is a device runtime for constrained edge devices optimized for minimal memory footprint (uses less than 15MB RAM). It has been introduced with AWS IoT Greengrass version 2.14.0 release and is backward compatible with AWS IoT Greengrass generic components, Greengrass V2 API, and SDK.

The Greengrass nucleus lite is offered as an alternative to the common Greengrass nucleus (`aws.greengrass.Nucleus`) and can be used in heterogeneous fleets of Greengrass devices.

Topics

- Versions
- Operating system
- Requirements
- Compatibility
- Download and installation
- Configuration
- Local log file
- Changelog

Versions

This component has the following versions:

- 2.5.x
- 2.4.x
- 2.3.x
- 2.2.x
- 2.1.x
- 2.0.x

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux (distributions with systemd)

For more information, see Greengrass nucleus.

Requirements

Devices must meet certain requirements to install and run the AWS IoT Greengrass nucleus lite and the AWS IoT Greengrass Core software. For more information, see Setup guide.

- 15MB of RAM space for the nucleus runtime.
- 5MB of storage (disk/FLASH).

Additional system dependencies are documented in the Setup Guide.

The Greengrass nucleus component is supported to run in a VPC. To deploy this component in a VPC, the following is required:

- The Greengrass nucleus must have connectivity to AWS IoT data, AWS IoT Credentials, and Amazon S3.

To run AWS IoT Greengrass nucleus lite as a non-root user, you can use a rootless container. For more information, see Using Podman in the AWS IoT Greengrass nucleus lite GitHub repository.

Compatibility

The AWS IoT Greengrass nucleus lite is compatible with the AWS IoT Greengrass v2 API (subset of) and supported SDKs. It does not depend on any specific language runtimes/VMs but components added to a deployment can require the presence of specific runtimes (e.g.: Java JVM, Python). For more information about what features are supported with Greengrass nucleus lite, see Greengrass feature compatibility.

Download and installation

You can download an apt package, build from source, use a Yocto layer, or download a pre-built Yocto image for compatible device (e.g., RaspberryPi). From the AWS IoT Core Console you will be able to download a **connection kit** containing all the credentials and initial configuration for your device. Instructions on how to install are included in each specific distribution method.

You can also follow a tutorial to install the AWS IoT Greengrass nucleus lite and explore Greengrass component development. For more information, see Tutorial: Getting started with AWS IoT Greengrass V2.

Configuration

The nucleus provides the following configuration parameters. Some parameters require that the AWS IoT Greengrass Core software restarts to take effect.

`iotRoleAlias`

The AWS IoT role alias that points to a token exchange IAM role. The AWS IoT credentials provider assumes this role to allow the Greengrass core device to interact with AWS services. For more information, see Authorize core devices to interact with AWS services.

`iotDataEndpoint`

The AWS IoT data endpoint for your AWS account.

`iotCredEndpoint`

The AWS IoT credentials endpoint for your AWS account.

`greengrassDataPlanePort`

The port to use for data plane connections. For more information, see Connect on port 443 or through a network proxy.

> **Important**
>
> You must specify a port where the device can make outbound connections. If you specify a port that is blocked, the device won't be able to connect to AWS IoT Greengrass to receive deployments. Choose from the following options:
>
> - `443`
> - `8443`
> - Default: `8443`

`awsRegion`

The AWS Region to use.

`runWithDefault`

The system user to use to run components.

> **Important**
>
> When you deploy a change to this configuration parameter, the AWS IoT Greengrass Core software restarts for the change to take effect.

This object contains the following information:

`posixUser`

The name or ID of the system user and, optionally, system group that the core device uses to run generic components. Specify the user and group separated by a colon (`:`) in the following format: `user:group`. The group is optional. If you don't specify a group, the AWS IoT Greengrass Core software uses the primary group for the user. For example, you can specify `ggc_user` or `ggc_user:ggc_group`. For more information, see Configure the user that runs components.

`networkProxy`

(Optional) The network proxy to use for all connections. For more information, see Connect on port 443 or through a network proxy.

> **Important**
>
> When you deploy a change to this configuration parameter, the change will take effect after the next restart of the AWS IoT Greengrass Core software.

This object contains the following information:

`noProxyAddresses`

(Optional) A comma-separated list of IP addresses or hostnames that are exempt from the proxy.

`proxy`

The proxy to which to connect. This object contains the following information:

`url`

The URL of the proxy server in the format `http://host:port`.

- `scheme` – The scheme, which must be `http`.
- `host` – The host name or IP address of the proxy server.
- `port` – (Optional) The port number. If you don't specify the port, then the Greengrass core device uses the following default value:
  - `http` – 80

Local log file

Messages are logged to stdout and log files are handled by systemd.

To view this component's logs

- Use `journalctl` to view logs.

Changelog

Version	Changes
2.5.1	Bug fixes and improvements • Fixes an issue where `SubscribeToConfigurationUpdate` could notify subscribers when a configuration value did not actually change. • Fixes an issue where the deployment source ARN of components could be overwritten by unrelated deployments. • Fixes an issue where HTTP artifact download retries did not trigger on retryable error codes such as 5xx and 429.
2.5.0	New features • Adds PKCS#11 support. You can use PKCS#11-backed key and certificate handles with OpenSSL Providers. • Adds support for using TPM with fleet provisioning. Bug fixes and improvements • Fixes a fleet provisioning regression introduced in v2.4.0.

Version	Changes
	• Fixes an artifact permissions ownership issue introduced in v2.4.0.
2.4.0	New features • Adds support for local deployment `--group-name` and `--remove-component` options. • Applies artifact permissions from recipes. Bug fixes and improvements • Fixes an issue where MQTT messages could be delayed until the next keepalive interval. • Fixes an issue where iotcored might enter deadlock and be unable to reconnect. • Fixes SSL verification failures when using an HTTP proxy. • Improves connection reliability.
2.3.3	Bug fixes and improvements • Fixes leak of file descriptors when creating components during deployment. • Fixes an issue where deployment job status reports might be rejected. • Allows component install phases to retry up to 3 times before failing deployment. • General bug fixes and improvements.
2.3.2	Bug fixes and improvements • Updates the version file for correct nucleus version reporting.
2.3.1	Bug fixes and improvements • GG will not attempt to update deployment state for canceled jobs. • Ensure DIR from fdopendir is closed. • Fleet provisioning will now trigger certificate file overwrite on each run. • Other minor bug fixes.

Version	Changes
2.3.0	New features • Support for using TPM 2.0 for IoT Core MQTT authorization. • The sample apt packages now support more operating systems: Ubuntu 22.04, Ubuntu 24.04, Debian 12, and Debian 13. • RestartComponent IPC is now supported. Bug fixes and improvements • Local deployments no longer need internet access. • GetConfiguration has been updated to match the Greengrass Nucleus runtime behavior. (**Breaking Change**) • General bug fixes and improvements.
2.2.2	Bug fixes and improvements • Fixes revised deployments containing unchanged component versions failing when an unchanged component had a running executable as an artifact. • Fixes recipe variable interpolation when the interpolated value has more than four nested subobjects. • Fixes recipe variable interpolation including quotes and shell special characters. • Fixes leak of fds when MQTT connections fail.
2.2.1	Bug fixes and improvements • Fixes an issue where the nucleus fails to obtain TES credentials.
2.2.0	New features • Adds support for container image artifact URIs. Bug fixes and improvements • General bug fixes and improvements.

Version	Changes
2.1.0	New features • Adds HTTP proxy support that can be configured using the `networkProxy` configuration option. Bug fixes and improvements • Lowers the requirement of `libcurl` from 7.86 to 7.82 to accommoda te devices running older `libcurl` versions. • Updates `journalctl` logs so they are attributed to components instead of recipe-runner. • Improves error responses for IPC calls. • Adds retries for S3 download attempts for generic component artifacts . • Minor bug fixes.
2.0.2	Bug fixes and improvements • Fixes dependencies of the apt packages to include `cgroup-tools` .
2.0.1	Bug fixes and improvements • Adds recipe variable interpolation support for Greengrass recipe's timeout section. • Adds support for ValidateAuthorizationToken IPC command for stream manager. • Fixes warnings from Fleet provisioning. • Adds retry and backoff to jobs listener. • General bug fixes and improvements.
2.0.0	Initial version.

Client device auth

The client device auth component (`aws.greengrass.clientdevices.Auth`) authenticates client devices and authorizes client device actions.

> **Note**
>
> Client devices are local IoT devices that connect to a Greengrass core device to send MQTT messages and data to process. For more information, see Interact with local IoT devices.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Local log file
- Changelog

Versions

> **Note**
>
> Client device auth version 2.3.0 has been discontinued. We strongly recommend that you upgrade to client device auth version 2.3.1 or later.

This component has the following versions:

- 2.5.x
- 2.4.x
- 2.3.x
- 2.2.x
- 2.1.x
- 2.0.x

Type

This component is a plugin component (`aws.greengrass.plugin`). The Greengrass nucleus runs this component in the same Java Virtual Machine (JVM) as the nucleus. The nucleus restarts when you change this component's version on the core device.

This component uses the same log file as the Greengrass nucleus. For more information, see Monitor AWS IoT Greengrass logs.

For more information, see Component types.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

Requirements

This component has the following requirements:

- The Greengrass service role must be associated to your AWS account and allow the `iot:DescribeCertificate` permission.
- The core device's AWS IoT policy must allow the following permissions:
  - `greengrass:GetConnectivityInfo`, where the resources include the ARN of the core device that runs this component
  - `greengrass:VerifyClientDeviceIoTCertificateAssociation`, where the resources include the Amazon Resource Name (ARN) of each client device that connects to the core device
  - `greengrass:VerifyClientDeviceIdentity`
  - `greengrass:PutCertificateAuthorities`
  - `iot:Publish`, where the resources include the ARN of the following MQTT topic:
    - `$aws/things/`*`coreDeviceThingName`*`*-gci/shadow/get`
  - `iot:Subscribe`, where the resources include the ARNs of the following MQTT topic filters:
    - `$aws/things/`*`coreDeviceThingName`*`*-gci/shadow/update/delta`

    - `$aws/things/coreDeviceThingName*-gci/shadow/get/accepted`
  - `iot:Receive`, where the resources include the ARNs of the following MQTT topics:
    - `$aws/things/coreDeviceThingName*-gci/shadow/update/delta`
    - `$aws/things/coreDeviceThingName*-gci/shadow/get/accepted`

  For more information, see AWS IoT policies for data plane operations and Minimal AWS IoT policy to support client devices.
- (Optional) To use offline authentication, the AWS Identity and Access Management (IAM) role used by the AWS IoT Greengrass service must contain the following permission:
  - `greengrass:ListClientDevicesAssociatedWithCoreDevice` to enable the core device to list clients for offline authentication.
- The client device auth component is supported to run in a VPC. To deploy this component in a VPC, the following is required.
  - The client device auth component must have connectivity to AWS IoT data, AWS IoT Credentials, and Amazon S3.

Endpoints and ports

This component must be able to perform outbound requests to the following endpoints and ports, in addition to endpoints and ports required for basic operation. For more information, see Allow device traffic through a proxy or firewall.

Endpoint	Port	Required	Description
`iot.region.amazonaws.com`	443	Yes	Used to get information about AWS IoT thing certificates.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

2.5.5

The following table lists the dependencies for version 2.5.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.6.0 <2.17.0	Soft

2.5.4

The following table lists the dependencies for version 2.5.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.6.0 <2.16.0	Soft

2.5.2 – 2.5.3

The following table lists the dependencies for versions 2.5.2 and 2.5.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.6.0 <2.15.0	Soft

2.5.1

The following table lists the dependencies for version 2.5.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.6.0 <2.14.0	Soft

2.4.4 - 2.5.0

The following table lists the dependencies for version 2.4.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.6.0 <2.13.0	Soft

2.4.3

The following table lists the dependencies for version 2.4.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.6.0 <2.12.0	Soft

2.4.1 and 2.4.2

The following table lists the dependencies for version 2.4.1 and 2.4.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.6.0 <2.11.0	Soft

2.3.0 – 2.4.0

The following table lists the dependencies for versions 2.3.0 to 2.4.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.6.0 <2.10.0	Soft

2.3.0

The following table lists the dependencies for version 2.3.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.6.0 <2.10.0	Soft

2.2.3

The following table lists the dependencies for version 2.2.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.6.0 <=2.9.0	Soft

2.2.2

The following table lists the dependencies for version 2.2.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.6.0 <=2.8.0	Soft

2.2.1

The following table lists the dependencies for version 2.2.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.6.0 <2.8.0	Soft

2.2.0

The following table lists the dependencies for version 2.2.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.6.0 <2.7.0	Soft

2.1.0

The following table lists the dependencies for version 2.1.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.2.0 <2.7.0	Soft

2.0.4

The following table lists the dependencies for version 2.0.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.2.0 <2.6.0	Soft

2.0.2 and 2.0.3

The following table lists the dependencies for versions 2.0.2 and 2.0.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.2.0 <2.5.0	Soft

2.0.1

The following table lists the dependencies for version 2.0.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.2.0 <2.4.0	Soft

2.0.0

The following table lists the dependencies for version 2.0.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.2.0 <2.3.0	Soft

For more information about component dependencies, see the component recipe reference.

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

Note

The subscribe permission is evaluated during a client subscribe request to the local MQTT broker. If the client's existing subscribe permission is revoked, the client will no longer be able to subscribe to a topic. It will, however, continue to receive messages from any previously subscribed topics. To prevent this behavior, the local MQTT broker should be restarted after revoking subscribe permission to force reauthorization of clients.
For the MQTT 5 broker (EMQX) component, update the `restartIdentifier` configuration to restart the MQTT 5 broker.
For the MQTT 3.1.1 broker (Moquette) component, it restarts weekly by default when the server certificate changes forcing clients to reauthorize. You can force a restart either by changing the connectivity information (IP addresses) of the core device or by making a deployment to remove the broker component and then deploy it again later.

v2.5.0 – 2.5.4

`deviceGroups`

Device groups are groups of client devices that have permissions to connect and communicate with a core device. Use selection rules to identify groups of client devices, and define *client device authorization policies* that specify the permissions for each device group.

This object contains the following information:

`formatVersion`

The format version for this configuration object.

Choose from the following options:

- `2021-03-05`

`definitions`

The device groups for this core device. Each definition specifies a *selection rule* to evaluate if a client device is a member of the group. Each definition also specifies the permissions policy to apply to client devices that match the selection rule. If a client device is a member of multiple device groups, the device's permissions are comprised of each group's permissions policy.

This object contains the following information:

`groupNameKey`

The name of this device group. Replace *`groupNameKey`* with a name that helps you identify this device group.

This object contains the following information:

`selectionRule`

The query that specifies which client devices are members of this device group. When a client device connects, the core device evaluates this selection rule to determine if the client device is a member of this device group. If the client device is a member, the core device uses this device group's policy to authorize the client device's actions.

Each selection rule comprises at least one *selection rule clause*, which is a single expression query that can match client devices. Selection rules use the same query syntax as AWS IoT fleet indexing. For more information about selection rule syntax, see AWS IoT fleet indexing query syntax in the *AWS IoT Core Developer Guide*.

Use the * wildcard to match multiple client devices with one selection rule clause. You can use this wildcard at the beginning and end of the thing name to match client devices whose names start or end with the string that you specify. You can also use this wildcard to match all client devices.

> **Note**
>
> To select a value that contains a colon character (`:`), escape the colon with a backslash character (\). In formats such as JSON, you must escape backslash characters, so you enter two backslash characters before the colon character. For example, specify `thingName:MyTeam\\:ClientDevice1` to select a thing whose name is `MyTeam:ClientDevice1`.

You can specify the following selector:

- `thingName` – The name of a client device's AWS IoT thing.

Example selection rule

The following selection rule matches client devices whose names are `MyClientDevice1` or `MyClientDevice2`.

```
thingName: MyClientDevice1 OR thingName: MyClientDevice2
```

Example selection rule (use wildcards)

The following selection rule matches client devices whose names start with `MyClientDevice`.

```
thingName: MyClientDevice*
```

Example selection rule (use wildcards)

The following selection rule matches client devices whose names end with `MyClientDevice`.

```
thingName: *MyClientDevice
```

Example selection rule (match all devices)

The following selection rule matches all client devices.

```
thingName: *
```

`policyName`

The permissions policy that applies to client devices in this device group. Specify the name of a policy that you define in the `policies` object.

`policies`

The client device authorization policies for client devices that connect to the core device. Each authorization policy specifies a set of actions and the resources where a client device can perform those actions.

This object contains the following information:

`policyNameKey`

The name of this authorization policy. Replace *`policyNameKey`* with a name that helps you identify this authorization policy. You use this policy name to define which policy applies to a device group.

This object contains the following information:

`statementNameKey`

The name of this policy statement. Replace *`statementNameKey`* with a name that helps you identify this policy statement.

This object contains the following information:

`operations`

The list of operations to allow for the resources in this policy.

You can include any of the following operations:

- `mqtt:connect` – Grants permission to connect to the core device. Client devices must have this permission to connect to a core device.

  This operation supports the following resources:

  - `mqtt:clientId:`*`deviceClientId`* – Restrict access based on the client ID that a client device uses to connect to the core device's MQTT broker. Replace *`deviceClientId`* with the client ID to use.
- `mqtt:publish` – Grants permission to publish MQTT messages to topics.

This operation supports the following resources:

- `mqtt:topic:`*`mqttTopic`* – Restrict access based on the MQTT topic where a client device publishes a message. Replace *`mqttTopic`* with the topic to use.

  This resource doesn't support MQTT topic wildcards.

- `mqtt:subscribe` – Grants permission to subscribe to MQTT topic filters to receive messages.

  This operation supports the following resources:

  - `mqtt:topicfilter:`*`mqttTopicFilter`* – Restrict access based on the MQTT topics where a client device can subscribe to messages. Replace *`mqttTopicFilter`* with the topic filter to use.

    This resource doesn't support MQTT topic wildcards.

`resources`

The list of resources to allow for the operations in this policy. Specify resources that correspond to the operations in this policy. For example, you might specify a list of MQTT topic resources (`mqtt:topic:`*`mqttTopic`*) in a policy that specifies the `mqtt:publish` operation.

You can specify the `*` wildcard anywhere within the resource variable to allow access to all resources. For example, you can specify **`mqtt:topic:my*`** to allow access to resources that match that input.

The following resource variable is supported:

- `mqtt:topic:${iot:Connection.Thing.ThingName}`

  This resolves to the name of the thing in the AWS IoT Core registry for which the policy is being evaluated. AWS IoT Core uses the certificate the device presents when it authenticates to determine which thing to use to verify the connection. This policy variable is only available when a device connects over MQTT or MQTT over the WebSocket protocol.

`statementDescription`

(Optional) A description for this policy statement.

`certificates`

(Optional) The certificate configuration options for this core device. This object contains the following information:

`serverCertificateValiditySeconds`

(Optional) The amount of time (in seconds) after which the local MQTT server certificate expires. You can configure this option to customize how often client devices disconnect and reconnect to the core device.

This component rotates the local MQTT server certificate 24 hours before it expires. The MQTT broker, such as the Moquette MQTT broker component, generates a new certificate and restarts. When this happens, all client devices connected to this core device are disconnected. Client devices can reconnect to the core device after a short period of time.

Default: `604800` (7 days)

Minimum value: `172800` (2 days)

Maximum value: `864000` (10 days)

`performance`

(Optional) The performance configuration options for this core device. This object contains the following information:

`maxActiveAuthTokens`

(Optional) The maximum number of active client device authorization tokens. You can increase this number to enable a greater number of client devices to connect to a single core device, without reauthenticating them.

Default: `2500`

`cloudRequestQueueSize`

(Optional) The maximum number of AWS Cloud requests to queue before this component rejects requests.

Default: `100`

`maxConcurrentCloudRequests`

(Optional) The maximum number of concurrent requests to send to the AWS Cloud. You can increase this number to improve authentication performance on core devices where you connect large numbers of client devices.

Default: 1

`certificateAuthority`

(Optional) Certificate authority configuration options to replace the core device intermediate authority with your own intermediate certificate authority.

> **Note**
>
> If you configure your Greengrass core device with a custom certificate authority (CA) and use the same CA to issue client device certificates, Greengrass bypasses authorization policy checks for client device MQTT operations. The client device auth component fully trusts clients using certificates signed by the CA that it is configured to use.
> To restrict this behavior when using a custom CA, create and sign client devices using a different CA or intermediate CA, then adjust the `certificateUri` and `certificateChainUri` fields to point to the correct intermediate CA.

This object contains the following information.

certificateUri

The location of the certificate. It can be a file system URI or a URI that points to a certificate stored in a hardware security module.

`certificateChainUri`

The location of the certificate chain for the core device CA. This should be the complete certificate chain back to your root CA. It can be a file system URI or a URI that points to a certificate chain stored in a hardware security module.

`privateKeyUri`

The location of the core device's private key. This can be a file system URI or a URI that points to a certificate private key stored in a hardware security module.

`security`

(Optional) Security configuration options for this core device. This object contains the following information.

`clientDeviceTrustDurationMinutes`

The duration in minutes that the authentication information of a client device can be trusted before it's required to reauthenticate with the core device. The default value is 1.

`metrics`

(Optional) The metrics options for this core device. Error metrics will only display if there is an error with the client device auth. This object contains the following information:

`disableMetrics`

If the `disableMetrics` field is set as `true`, the client device auth won't collect metrics.

Default: `false`

`aggregatePeriodSeconds`

The aggregation period in seconds that determines how often the client device auth aggregates metrics and sends them to the telemetry agent. This doesn't change how often metrics are published because the telemetry agent still publishes them once a day.

Default: `3600`

startupTimeoutSeconds

(Optional) The maximum of time in seconds for the component to start. The component's state changes to `ERRORED` if it exceeds this timeout.

Default: `120`

Example: Configuration merge update (using a restrictive policy)

The following example configuration specifies to allow client devices whose names start with `MyClientDevice` to connect and publish/subscribe on all topics.

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
```

```
    "definitions": {
      "MyDeviceGroup": {
        "selectionRule": "thingName: MyClientDevice*",
        "policyName": "MyRestrictivePolicy"
      }
    },
    "policies": {
      "MyRestrictivePolicy": {
        "AllowConnect": {
          "statementDescription": "Allow client devices to connect.",
          "operations": [
            "mqtt:connect"
          ],
          "resources": [
            "*"
          ]
        },
        "AllowPublish": {
          "statementDescription": "Allow client devices to publish on test/topic.",
          "operations": [
            "mqtt:publish"
          ],
          "resources": [
            "mqtt:topic:test/topic"
          ]
        },
        "AllowSubscribe": {
          "statementDescription": "Allow client devices to subscribe to test/topic/
response.",
          "operations": [
            "mqtt:subscribe"
          ],
          "resources": [
            "mqtt:topicfilter:test/topic/response"
          ]
        }
      }
    }
  }
}
```

Example: Configuration merge update (using a permissive policy)

The following example configuration specifies to allow all client devices to connect and publish/subscribe on all topics.

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyPermissiveDeviceGroup": {
        "selectionRule": "thingName: *",
        "policyName": "MyPermissivePolicy"
      }
    },
    "policies": {
      "MyPermissivePolicy": {
        "AllowAll": {
          "statementDescription": "Allow client devices to perform all actions.",
          "operations": [
            "*"
          ],
          "resources": [
            "*"
          ]
        }
      }
    }
  }
}
```

Example: Configuration merge update (using a thing name policy)

The following example configuration enables client devices to publish on topics that begin with the client device's thing name and end with the string `topic`.

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "myThing": {
        "selectionRule": "thingName: *",
        "policyName": "MyThingNamePolicy"
      }
```

```
    },
    "policies": {
      "MyThingNamePolicy": {
        "policyStatement": {
          "statementDescription": "mqtt publish",
          "operations": [
            "mqtt:publish"
          ],
          "resources": [
            "mqtt:topic:${iot:Connection.Thing.ThingName}/*/topic"
          ]
        }
      }
    }
  }
}
```

v2.4.5

`deviceGroups`

Device groups are groups of client devices that have permissions to connect and communicate with a core device. Use selection rules to identify groups of client devices, and define *client device authorization policies* that specify the permissions for each device group.

This object contains the following information:

`formatVersion`

The format version for this configuration object.

Choose from the following options:

- `2021-03-05`

`definitions`

The device groups for this core device. Each definition specifies a *selection rule* to evaluate if a client device is a member of the group. Each definition also specifies the permissions policy to apply to client devices that match the selection rule. If a client device is a member of multiple device groups, the device's permissions are comprised of each group's permissions policy.

This object contains the following information:

groupNameKey

The name of this device group. Replace *groupNameKey* with a name that helps you identify this device group.

This object contains the following information:

`selectionRule`

The query that specifies which client devices are members of this device group. When a client device connects, the core device evaluates this selection rule to determine if the client device is a member of this device group. If the client device is a member, the core device uses this device group's policy to authorize the client device's actions.

Each selection rule comprises at least one *selection rule clause*, which is a single expression query that can match client devices. Selection rules use the same query syntax as AWS IoT fleet indexing. For more information about selection rule syntax, see AWS IoT fleet indexing query syntax in the *AWS IoT Core Developer Guide*.

Use the * wildcard to match multiple client devices with one selection rule clause. You can use this wildcard at the beginning and end of the thing name to match client devices whose names start or end with the string that you specify. You can also use this wildcard to match all client devices.

> **Note**
>
> To select a value that contains a colon character (`:`), escape the colon with a backslash character (\). In formats such as JSON, you must escape backslash characters, so you enter two backslash characters before the colon character. For example, specify `thingName:MyTeam\\:ClientDevice1` to select a thing whose name is `MyTeam:ClientDevice1`.

You can specify the following selector:

- `thingName` – The name of a client device's AWS IoT thing.

Example selection rule

The following selection rule matches client devices whose names are `MyClientDevice1` or `MyClientDevice2`.

```
thingName: MyClientDevice1 OR thingName: MyClientDevice2
```

Example selection rule (use wildcards)

The following selection rule matches client devices whose names start with `MyClientDevice`.

```
thingName: MyClientDevice*
```

Example selection rule (use wildcards)

The following selection rule matches client devices whose names end with `MyClientDevice`.

```
thingName: *MyClientDevice
```

Example selection rule (match all devices)

The following selection rule matches all client devices.

```
thingName: *
```

`policyName`

The permissions policy that applies to client devices in this device group. Specify the name of a policy that you define in the `policies` object.

`policies`

The client device authorization policies for client devices that connect to the core device. Each authorization policy specifies a set of actions and the resources where a client device can perform those actions.

This object contains the following information:

policyNameKey

The name of this authorization policy. Replace *policyNameKey* with a name that helps you identify this authorization policy. You use this policy name to define which policy applies to a device group.

This object contains the following information:

statementNameKey

The name of this policy statement. Replace *statementNameKey* with a name that helps you identify this policy statement.

This object contains the following information:

`operations`

The list of operations to allow for the resources in this policy.

You can include any of the following operations:

- `mqtt:connect` – Grants permission to connect to the core device. Client devices must have this permission to connect to a core device.

  This operation supports the following resources:

  - `mqtt:clientId:`*deviceClientId* – Restrict access based on the client ID that a client device uses to connect to the core device's MQTT broker. Replace *deviceClientId* with the client ID to use.
- `mqtt:publish` – Grants permission to publish MQTT messages to topics.

  This operation supports the following resources:

  - `mqtt:topic:`*mqttTopic* – Restrict access based on the MQTT topic where a client device publishes a message. Replace *mqttTopic* with the topic to use.

    This resource doesn't support MQTT topic wildcards.
- `mqtt:subscribe` – Grants permission to subscribe to MQTT topic filters to receive messages.

  This operation supports the following resources:

- `mqtt:topicfilter:`*`mqttTopicFilter`* – Restrict access based on the MQTT topics where a client device can subscribe to messages. Replace *`mqttTopicFilter`* with the topic filter to use.

  This resource supports the + and # MQTT topic wildcards. For more information, see MQTT topics in the *AWS IoT Core Developer Guide*.

  The client device can subscribe to the exact topic filters that you allow. For example, if you allow the client device to subscribe to the `mqtt:topicfilter:client/+/status` resource, the client device can subscribe to `client/+/status` but not `client/client1/status`.

You can specify the * wildcard to allow access to all actions.

`resources`

The list of resources to allow for the operations in this policy. Specify resources that correspond to the operations in this policy. For example, you might specify a list of MQTT topic resources (`mqtt:topic:`*`mqttTopic`*) in a policy that specifies the `mqtt:publish` operation.

You can specify the * wildcard to allow access to all resources. You can't use the * wildcard to match partial resource identifiers. For example, you can specify **`"resources": "*"`**, but you can't specify **`"resources": "mqtt:clientId:*"`**.

`statementDescription`

(Optional) A description for this policy statement.

`certificates`

(Optional) The certificate configuration options for this core device. This object contains the following information:

`serverCertificateValiditySeconds`

(Optional) The amount of time (in seconds) after which the local MQTT server certificate expires. You can configure this option to customize how often client devices disconnect and reconnect to the core device.

This component rotates the local MQTT server certificate 24 hours before it expires. The MQTT broker, such as the Moquette MQTT broker component, generates a new

certificate and restarts. When this happens, all client devices connected to this core device are disconnected. Client devices can reconnect to the core device after a short period of time.

Default: `604800` (7 days)

Minimum value: `172800` (2 days)

Maximum value: `864000` (10 days)

`performance`

(Optional) The performance configuration options for this core device. This object contains the following information:

`maxActiveAuthTokens`

(Optional) The maximum number of active client device authorization tokens. You can increase this number to enable a greater number of client devices to connect to a single core device, without reauthenticating them.

Default: `2500`

`cloudRequestQueueSize`

(Optional) The maximum number of AWS Cloud requests to queue before this component rejects requests.

Default: `100`

`maxConcurrentCloudRequests`

(Optional) The maximum number of concurrent requests to send to the AWS Cloud. You can increase this number to improve authentication performance on core devices where you connect large numbers of client devices.

Default: 1

`certificateAuthority`

(Optional) Certificate authority configuration options to replace the core device intermediate authority with your own intermediate certificate authority.

> **Note**
>
> If you configure your Greengrass core device with a custom certificate authority (CA) and use the same CA to issue client device certificates, Greengrass bypasses authorization policy checks for client device MQTT operations. The client device auth component fully trusts clients using certificates signed by the CA that it is configured to use.
> To restrict this behavior when using a custom CA, create and sign client devices using a different CA or intermediate CA, then adjust the `certificateUri` and `certificateChainUri` fields to point to the correct intermediate CA.

This object contains the following information.

certificateUri

The location of the certificate. It can be a file system URI or a URI that points to a certificate stored in a hardware security module.

`certificateChainUri`

The location of the certificate chain for the core device CA. This should be the complete certificate chain back to your root CA. It can be a file system URI or a URI that points to a certificate chain stored in a hardware security module.

`privateKeyUri`

The location of the core device's private key. This can be a file system URI or a URI that points to a certificate private key stored in a hardware security module.

`security`

(Optional) Security configuration options for this core device. This object contains the following information.

`clientDeviceTrustDurationMinutes`

The duration in minutes that the authentication information of a client device can be trusted before it's required to reauthenticate with the core device. The default value is 1.

`metrics`

(Optional) The metrics options for this core device. Error metrics will only display if there is an error with the client device auth. This object contains the following information:

`disableMetrics`

If the `disableMetrics` field is set as `true`, the client device auth won't collect metrics.

Default: `false`

`aggregatePeriodSeconds`

The aggregation period in seconds that determines how often the client device auth aggregates metrics and sends them to the telemetry agent. This doesn't change how often metrics are published because the telemetry agent still publishes them once a day.

Default: `3600`

startupTimeoutSeconds

(Optional) The maximum of time in seconds for the component to start. The component's state changes to `ERRORED` if it exceeds this timeout.

Default: `120`

Example: Configuration merge update (using a restrictive policy)

The following example configuration specifies to allow client devices whose names start with `MyClientDevice` to connect and publish/subscribe on all topics.

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyDeviceGroup": {
        "selectionRule": "thingName: MyClientDevice*",
        "policyName": "MyRestrictivePolicy"
      }
    },
    "policies": {
      "MyRestrictivePolicy": {
        "AllowConnect": {
```

```
          "statementDescription": "Allow client devices to connect.",
          "operations": [
            "mqtt:connect"
          ],
          "resources": [
            "*"
          ]
        },
        "AllowPublish": {
          "statementDescription": "Allow client devices to publish on test/topic.",
          "operations": [
            "mqtt:publish"
          ],
          "resources": [
            "mqtt:topic:test/topic"
          ]
        },
        "AllowSubscribe": {
          "statementDescription": "Allow client devices to subscribe to test/topic/
response.",
          "operations": [
            "mqtt:subscribe"
          ],
          "resources": [
            "mqtt:topicfilter:test/topic/response"
          ]
        }
      }
    }
  }
}
```

Example: Configuration merge update (using a permissive policy)

The following example configuration specifies to allow all client devices to connect and publish/subscribe on all topics.

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyPermissiveDeviceGroup": {
        "selectionRule": "thingName: *",
```

```
        "policyName": "MyPermissivePolicy"
      }
    },
    "policies": {
      "MyPermissivePolicy": {
        "AllowAll": {
          "statementDescription": "Allow client devices to perform all actions.",
          "operations": [
            "*"
          ],
          "resources": [
            "*"
          ]
        }
      }
    }
  }
}
```

v2.4.2 - v2.4.4

`deviceGroups`

Device groups are groups of client devices that have permissions to connect and communicate with a core device. Use selection rules to identify groups of client devices, and define *client device authorization policies* that specify the permissions for each device group.

This object contains the following information:

`formatVersion`

The format version for this configuration object.

Choose from the following options:

- `2021-03-05`

`definitions`

The device groups for this core device. Each definition specifies a *selection rule* to evaluate if a client device is a member of the group. Each definition also specifies the permissions policy to apply to client devices that match the selection rule. If a client device is a member of multiple device groups, the device's permissions are comprised of each group's permissions policy.

This object contains the following information:

groupNameKey

The name of this device group. Replace *groupNameKey* with a name that helps you identify this device group.

This object contains the following information:

`selectionRule`

The query that specifies which client devices are members of this device group. When a client device connects, the core device evaluates this selection rule to determine if the client device is a member of this device group. If the client device is a member, the core device uses this device group's policy to authorize the client device's actions.

Each selection rule comprises at least one *selection rule clause*, which is a single expression query that can match client devices. Selection rules use the same query syntax as AWS IoT fleet indexing. For more information about selection rule syntax, see AWS IoT fleet indexing query syntax in the *AWS IoT Core Developer Guide*.

Use the `*` wildcard to match multiple client devices with one selection rule clause. You can use this wildcard at the end of the thing name to match client devices whose names start with a string that you specify. You can also use this wildcard to match all client devices.

> **Note**
>
> To select a value that contains a colon character (`:`), escape the colon with a backslash character (`\\`). In formats such as JSON, you must escape backslash characters, so you enter two backslash characters before the colon character. For example, specify `thingName:MyTeam\\\\:ClientDevice1` to select a thing whose name is `MyTeam:ClientDevice1`.

You can specify the following selector:

- `thingName` – The name of a client device's AWS IoT thing.

Example selection rule

The following selection rule matches client devices whose names are `MyClientDevice1` or `MyClientDevice2`.

```
thingName: MyClientDevice1 OR thingName: MyClientDevice2
```

Example selection rule (use wildcards)

The following selection rule matches client devices whose names start with `MyClientDevice`.

```
thingName: MyClientDevice*
```

Example selection rule (match all devices)

The following selection rule matches all client devices.

```
thingName: *
```

`policyName`

The permissions policy that applies to client devices in this device group. Specify the name of a policy that you define in the `policies` object.

`policies`

The client device authorization policies for client devices that connect to the core device. Each authorization policy specifies a set of actions and the resources where a client device can perform those actions.

This object contains the following information:

`policyNameKey`

The name of this authorization policy. Replace *`policyNameKey`* with a name that helps you identify this authorization policy. You use this policy name to define which policy applies to a device group.

This object contains the following information:

statementNameKey

The name of this policy statement. Replace *statementNameKey* with a name that helps you identify this policy statement.

This object contains the following information:

`operations`

The list of operations to allow for the resources in this policy.

You can include any of the following operations:

- `mqtt:connect` – Grants permission to connect to the core device. Client devices must have this permission to connect to a core device.

  This operation supports the following resources:

  - `mqtt:clientId:`*deviceClientId* – Restrict access based on the client ID that a client device uses to connect to the core device's MQTT broker. Replace *deviceClientId* with the client ID to use.
- `mqtt:publish` – Grants permission to publish MQTT messages to topics.

  This operation supports the following resources:

  - `mqtt:topic:`*mqttTopic* – Restrict access based on the MQTT topic where a client device publishes a message. Replace *mqttTopic* with the topic to use.

    This resource doesn't support MQTT topic wildcards.
- `mqtt:subscribe` – Grants permission to subscribe to MQTT topic filters to receive messages.

  This operation supports the following resources:

  - `mqtt:topicfilter:`*mqttTopicFilter* – Restrict access based on the MQTT topics where a client device can subscribe to messages. Replace *mqttTopicFilter* with the topic filter to use.

    This resource supports the + and # MQTT topic wildcards. For more information, see MQTT topics in the *AWS IoT Core Developer Guide*.

    The client device can subscribe to the exact topic filters that you allow. For example, if you allow the client device to subscribe to the

`mqtt:topicfilter:client/+/status` resource, the client device can subscribe to `client/+/status` but not `client/client1/status`.

You can specify the `*` wildcard to allow access to all actions.

`resources`

The list of resources to allow for the operations in this policy. Specify resources that correspond to the operations in this policy. For example, you might specify a list of MQTT topic resources (`mqtt:topic:`*`mqttTopic`*) in a policy that specifies the `mqtt:publish` operation.

You can specify the `*` wildcard to allow access to all resources. You can't use the `*` wildcard to match partial resource identifiers. For example, you can specify **`"resources": "*"`**, but you can't specify **`"resources": "mqtt:clientId:*"`**.

`statementDescription`

(Optional) A description for this policy statement.

`certificates`

(Optional) The certificate configuration options for this core device. This object contains the following information:

`serverCertificateValiditySeconds`

(Optional) The amount of time (in seconds) after which the local MQTT server certificate expires. You can configure this option to customize how often client devices disconnect and reconnect to the core device.

This component rotates the local MQTT server certificate 24 hours before it expires. The MQTT broker, such as the Moquette MQTT broker component, generates a new certificate and restarts. When this happens, all client devices connected to this core device are disconnected. Client devices can reconnect to the core device after a short period of time.

Default: `604800` (7 days)

Minimum value: `172800` (2 days)

Maximum value: `864000` (10 days)

`performance`

(Optional) The performance configuration options for this core device. This object contains the following information:

`maxActiveAuthTokens`

(Optional) The maximum number of active client device authorization tokens. You can increase this number to enable a greater number of client devices to connect to a single core device, without reauthenticating them.

Default: `2500`

`cloudRequestQueueSize`

(Optional) The maximum number of AWS Cloud requests to queue before this component rejects requests.

Default: `100`

`maxConcurrentCloudRequests`

(Optional) The maximum number of concurrent requests to send to the AWS Cloud. You can increase this number to improve authentication performance on core devices where you connect large numbers of client devices.

Default: 1

`certificateAuthority`

(Optional) Certificate authority configuration options to replace the core device intermediate authority with your own intermediate certificate authority.

> **Note**
>
> If you configure your Greengrass core device with a custom certificate authority (CA) and use the same CA to issue client device certificates, Greengrass bypasses authorization policy checks for client device MQTT operations. The client device auth component fully trusts clients using certificates signed by the CA that it is configured to use.
> To restrict this behavior when using a custom CA, create and sign client devices using a different CA or intermediate CA, then adjust the `certificateUri` and `certificateChainUri` fields to point to the correct intermediate CA.

This object contains the following information.

certificateUri

The location of the certificate. It can be a file system URI or a URI that points to a certificate stored in a hardware security module.

`certificateChainUri`

The location of the certificate chain for the core device CA. This should be the complete certificate chain back to your root CA. It can be a file system URI or a URI that points to a certificate chain stored in a hardware security module.

`privateKeyUri`

The location of the core device's private key. This can be a file system URI or a URI that points to a certificate private key stored in a hardware security module.

`security`

(Optional) Security configuration options for this core device. This object contains the following information.

`clientDeviceTrustDurationMinutes`

The duration in minutes that the authentication information of a client device can be trusted before it's required to reauthenticate with the core device. The default value is 1.

`metrics`

(Optional) The metrics options for this core device. Error metrics will only display if there is an error with the client device auth. This object contains the following information:

`disableMetrics`

If the `disableMetrics` field is set as `true`, the client device auth won't collect metrics.

Default: `false`

`aggregatePeriodSeconds`

The aggregation period in seconds that determines how often the client device auth aggregates metrics and sends them to the telemetry agent. This doesn't change how often metrics are published because the telemetry agent still publishes them once a day.

Default: `3600`

startupTimeoutSeconds

(Optional) The maximum of time in seconds for the component to start. The component's state changes to `ERRORED` if it exceeds this timeout.

Default: `120`

Example: Configuration merge update (using a restrictive policy)

The following example configuration specifies to allow client devices whose names start with `MyClientDevice` to connect and publish/subscribe on all topics.

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyDeviceGroup": {
        "selectionRule": "thingName: MyClientDevice*",
        "policyName": "MyRestrictivePolicy"
      }
    },
    "policies": {
      "MyRestrictivePolicy": {
        "AllowConnect": {
          "statementDescription": "Allow client devices to connect.",
          "operations": [
            "mqtt:connect"
          ],
          "resources": [
            "*"
          ]
        },
        "AllowPublish": {
          "statementDescription": "Allow client devices to publish on test/topic.",
          "operations": [
            "mqtt:publish"
          ],
          "resources": [
            "mqtt:topic:test/topic"
          ]
        },
```

```
        "AllowSubscribe": {
          "statementDescription": "Allow client devices to subscribe to test/topic/
response.",
          "operations": [
            "mqtt:subscribe"
          ],
          "resources": [
            "mqtt:topicfilter:test/topic/response"
          ]
        }
      }
    }
  }
}
```

Example: Configuration merge update (using a permissive policy)

The following example configuration specifies to allow all client devices to connect and publish/subscribe on all topics.

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyPermissiveDeviceGroup": {
        "selectionRule": "thingName: *",
        "policyName": "MyPermissivePolicy"
      }
    },
    "policies": {
      "MyPermissivePolicy": {
        "AllowAll": {
          "statementDescription": "Allow client devices to perform all actions.",
          "operations": [
            "*"
          ],
          "resources": [
            "*"
          ]
        }
      }
    }
  }
```

```
}
```

v2.4.0 - v2.4.1

`deviceGroups`

Device groups are groups of client devices that have permissions to connect and communicate with a core device. Use selection rules to identify groups of client devices, and define *client device authorization policies* that specify the permissions for each device group.

This object contains the following information:

`formatVersion`

The format version for this configuration object.

Choose from the following options:

- `2021-03-05`

`definitions`

The device groups for this core device. Each definition specifies a *selection rule* to evaluate if a client device is a member of the group. Each definition also specifies the permissions policy to apply to client devices that match the selection rule. If a client device is a member of multiple device groups, the device's permissions are comprised of each group's permissions policy.

This object contains the following information:

`groupNameKey`

The name of this device group. Replace *`groupNameKey`* with a name that helps you identify this device group.

This object contains the following information:

`selectionRule`

The query that specifies which client devices are members of this device group. When a client device connects, the core device evaluates this selection rule to determine if the client device is a member of this device group. If the client device

is a member, the core device uses this device group's policy to authorize the client device's actions.

Each selection rule comprises at least one *selection rule clause*, which is a single expression query that can match client devices. Selection rules use the same query syntax as AWS IoT fleet indexing. For more information about selection rule syntax, see AWS IoT fleet indexing query syntax in the *AWS IoT Core Developer Guide*.

Use the `*` wildcard to match multiple client devices with one selection rule clause. You can use this wildcard at the end of the thing name to match client devices whose names start with a string that you specify. You can also use this wildcard to match all client devices.

> **Note**
>
> To select a value that contains a colon character (`:`), escape the colon with a backslash character (`\\`). In formats such as JSON, you must escape backslash characters, so you enter two backslash characters before the colon character. For example, specify `thingName:MyTeam\\\\:ClientDevice1` to select a thing whose name is `MyTeam:ClientDevice1`.

You can specify the following selector:

- `thingName` – The name of a client device's AWS IoT thing.

Example selection rule

The following selection rule matches client devices whose names are `MyClientDevice1` or `MyClientDevice2`.

```
thingName: MyClientDevice1 OR thingName: MyClientDevice2
```

Example selection rule (use wildcards)

The following selection rule matches client devices whose names start with `MyClientDevice`.

```
thingName: MyClientDevice*
```

Example selection rule (match all devices)

The following selection rule matches all client devices.

```
thingName: *
```

`policyName`

The permissions policy that applies to client devices in this device group. Specify the name of a policy that you define in the `policies` object.

`policies`

The client device authorization policies for client devices that connect to the core device. Each authorization policy specifies a set of actions and the resources where a client device can perform those actions.

This object contains the following information:

`policyNameKey`

The name of this authorization policy. Replace *`policyNameKey`* with a name that helps you identify this authorization policy. You use this policy name to define which policy applies to a device group.

This object contains the following information:

`statementNameKey`

The name of this policy statement. Replace *`statementNameKey`* with a name that helps you identify this policy statement.

This object contains the following information:

`operations`

The list of operations to allow for the resources in this policy.

You can include any of the following operations:

- `mqtt:connect` – Grants permission to connect to the core device. Client devices must have this permission to connect to a core device.

  This operation supports the following resources:

  - `mqtt:clientId:deviceClientId` – Restrict access based on the client ID that a client device uses to connect to the core device's MQTT broker. Replace *`deviceClientId`* with the client ID to use.
- `mqtt:publish` – Grants permission to publish MQTT messages to topics.

  This operation supports the following resources:

  - `mqtt:topic:mqttTopic` – Restrict access based on the MQTT topic where a client device publishes a message. Replace *`mqttTopic`* with the topic to use.

    This resource doesn't support MQTT topic wildcards.
- `mqtt:subscribe` – Grants permission to subscribe to MQTT topic filters to receive messages.

  This operation supports the following resources:

  - `mqtt:topicfilter:mqttTopicFilter` – Restrict access based on the MQTT topics where a client device can subscribe to messages. Replace *`mqttTopicFilter`* with the topic filter to use.

    This resource supports the + and # MQTT topic wildcards. For more information, see MQTT topics in the *AWS IoT Core Developer Guide*.

    The client device can subscribe to the exact topic filters that you allow. For example, if you allow the client device to subscribe to the `mqtt:topicfilter:client/+/status` resource, the client device can subscribe to `client/+/status` but not `client/client1/status`.

You can specify the * wildcard to allow access to all actions.

`resources`

The list of resources to allow for the operations in this policy. Specify resources that correspond to the operations in this policy. For example, you might specify a list of MQTT topic resources (`mqtt:topic:mqttTopic`) in a policy that specifies the `mqtt:publish` operation.

You can specify the * wildcard to allow access to all resources. You can't use the * wildcard to match partial resource identifiers. For example, you can specify **"resources": "*"**, but you can't specify **"resources": "mqtt:clientId:*"**.

`statementDescription`

(Optional) A description for this policy statement.

`certificates`

(Optional) The certificate configuration options for this core device. This object contains the following information:

`serverCertificateValiditySeconds`

(Optional) The amount of time (in seconds) after which the local MQTT server certificate expires. You can configure this option to customize how often client devices disconnect and reconnect to the core device.

This component rotates the local MQTT server certificate 24 hours before it expires. The MQTT broker, such as the Moquette MQTT broker component, generates a new certificate and restarts. When this happens, all client devices connected to this core device are disconnected. Client devices can reconnect to the core device after a short period of time.

Default: `604800` (7 days)

Minimum value: `172800` (2 days)

Maximum value: `864000` (10 days)

`performance`

(Optional) The performance configuration options for this core device. This object contains the following information:

`maxActiveAuthTokens`

(Optional) The maximum number of active client device authorization tokens. You can increase this number to enable a greater number of client devices to connect to a single core device, without reauthenticating them.

Default: `2500`

`cloudRequestQueueSize`

(Optional) The maximum number of AWS Cloud requests to queue before this component rejects requests.

Default: `100`

`maxConcurrentCloudRequests`

(Optional) The maximum number of concurrent requests to send to the AWS Cloud. You can increase this number to improve authentication performance on core devices where you connect large numbers of client devices.

Default: 1

`certificateAuthority`

(Optional) Certificate authority configuration options to replace the core device intermediate authority with your own intermediate certificate authority. This object contains the following information.

This object contains the following information:

certificateUri

The location of the certificate. It can be a file system URI or a URI that points to a certificate stored in a hardware security module.

`certificateChainUri`

The location of the certificate chain for the core device CA. This should be the complete certificate chain back to your root CA. It can be a file system URI or a URI that points to a certificate chain stored in a hardware security module.

`privateKeyUri`

The location of the core device's private key. This can be a file system URI or a URI that points to a certificate private key stored in a hardware security module.

`security`

(Optional) Security configuration options for this core device. This object contains the following information.

`clientDeviceTrustDurationMinutes`

The duration in minutes that the authentication information of a client device can be trusted before it's required to reauthenticate with the core device. The default value is 1.

`metrics`

(Optional) The metrics options for this core device. Error metrics will only display if there is an error with the client device auth. This object contains the following information:

`disableMetrics`

If the `disableMetrics` field is set as `true`, the client device auth won't collect metrics.

Default: `false`

`aggregatePeriodSeconds`

The aggregation period in seconds that determines how often the client device auth aggregates metrics and sends them to the telemetry agent. This doesn't change how often metrics are published because the telemetry agent still publishes them once a day.

Default: `3600`

Example: Configuration merge update (using a restrictive policy)

The following example configuration specifies to allow client devices whose names start with `MyClientDevice` to connect and publish/subscribe on all topics.

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyDeviceGroup": {
        "selectionRule": "thingName: MyClientDevice*",
        "policyName": "MyRestrictivePolicy"
      }
    },
    "policies": {
      "MyRestrictivePolicy": {
        "AllowConnect": {
          "statementDescription": "Allow client devices to connect.",
```

```
          "operations": [
            "mqtt:connect"
          ],
          "resources": [
            "*"
          ]
        },
        "AllowPublish": {
          "statementDescription": "Allow client devices to publish on test/topic.",
          "operations": [
            "mqtt:publish"
          ],
          "resources": [
            "mqtt:topic:test/topic"
          ]
        },
        "AllowSubscribe": {
          "statementDescription": "Allow client devices to subscribe to test/topic/
response.",
          "operations": [
            "mqtt:subscribe"
          ],
          "resources": [
            "mqtt:topicfilter:test/topic/response"
          ]
        }
      }
    }
  }
}
```

Example: Configuration merge update (using a permissive policy)

The following example configuration specifies to allow all client devices to connect and publish/subscribe on all topics.

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyPermissiveDeviceGroup": {
        "selectionRule": "thingName: *",
        "policyName": "MyPermissivePolicy"
```

```
      }
    },
    "policies": {
      "MyPermissivePolicy": {
        "AllowAll": {
          "statementDescription": "Allow client devices to perform all actions.",
          "operations": [
            "*"
          ],
          "resources": [
            "*"
          ]
        }
      }
    }
  }
}
```

v2.3.x

`deviceGroups`

Device groups are groups of client devices that have permissions to connect and communicate with a core device. Use selection rules to identify groups of client devices, and define *client device authorization policies* that specify the permissions for each device group.

This object contains the following information:

`formatVersion`

The format version for this configuration object.

Choose from the following options:

- `2021-03-05`

`definitions`

The device groups for this core device. Each definition specifies a *selection rule* to evaluate if a client device is a member of the group. Each definition also specifies the permissions policy to apply to client devices that match the selection rule. If a client device is a member of multiple device groups, the device's permissions are comprised of each group's permissions policy.

This object contains the following information:

groupNameKey

The name of this device group. Replace *groupNameKey* with a name that helps you identify this device group.

This object contains the following information:

`selectionRule`

The query that specifies which client devices are members of this device group. When a client device connects, the core device evaluates this selection rule to determine if the client device is a member of this device group. If the client device is a member, the core device uses this device group's policy to authorize the client device's actions.

Each selection rule comprises at least one *selection rule clause*, which is a single expression query that can match client devices. Selection rules use the same query syntax as AWS IoT fleet indexing. For more information about selection rule syntax, see AWS IoT fleet indexing query syntax in the *AWS IoT Core Developer Guide*.

Use the * wildcard to match multiple client devices with one selection rule clause. You can use this wildcard at the end of the thing name to match client devices whose names start with a string that you specify. You can also use this wildcard to match all client devices.

> **Note**
>
> To select a value that contains a colon character (`:`), escape the colon with a backslash character (\\). In formats such as JSON, you must escape backslash characters, so you enter two backslash characters before the colon character. For example, specify `thingName:MyTeam\\\\:ClientDevice1` to select a thing whose name is `MyTeam:ClientDevice1`.

You can specify the following selector:

- `thingName` – The name of a client device's AWS IoT thing.

Example selection rule

The following selection rule matches client devices whose names are `MyClientDevice1` or `MyClientDevice2`.

```
thingName: MyClientDevice1 OR thingName: MyClientDevice2
```

Example selection rule (use wildcards)

The following selection rule matches client devices whose names start with `MyClientDevice`.

```
thingName: MyClientDevice*
```

Example selection rule (match all devices)

The following selection rule matches all client devices.

```
thingName: *
```

`policyName`

The permissions policy that applies to client devices in this device group. Specify the name of a policy that you define in the `policies` object.

`policies`

The client device authorization policies for client devices that connect to the core device. Each authorization policy specifies a set of actions and the resources where a client device can perform those actions.

This object contains the following information:

`policyNameKey`

The name of this authorization policy. Replace *`policyNameKey`* with a name that helps you identify this authorization policy. You use this policy name to define which policy applies to a device group.

This object contains the following information:

statementNameKey

The name of this policy statement. Replace *statementNameKey* with a name that helps you identify this policy statement.

This object contains the following information:

`operations`

The list of operations to allow for the resources in this policy.

You can include any of the following operations:

- `mqtt:connect` – Grants permission to connect to the core device. Client devices must have this permission to connect to a core device.

  This operation supports the following resources:

  - `mqtt:clientId:`*deviceClientId* – Restrict access based on the client ID that a client device uses to connect to the core device's MQTT broker. Replace *deviceClientId* with the client ID to use.
- `mqtt:publish` – Grants permission to publish MQTT messages to topics.

  This operation supports the following resources:

  - `mqtt:topic:`*mqttTopic* – Restrict access based on the MQTT topic where a client device publishes a message. Replace *mqttTopic* with the topic to use.

    This resource doesn't support MQTT topic wildcards.
- `mqtt:subscribe` – Grants permission to subscribe to MQTT topic filters to receive messages.

  This operation supports the following resources:

  - `mqtt:topicfilter:`*mqttTopicFilter* – Restrict access based on the MQTT topics where a client device can subscribe to messages. Replace *mqttTopicFilter* with the topic filter to use.

    This resource supports the + and # MQTT topic wildcards. For more information, see MQTT topics in the *AWS IoT Core Developer Guide*.

    The client device can subscribe to the exact topic filters that you allow. For example, if you allow the client device to subscribe to the

`mqtt:topicfilter:client/+/status` resource, the client device can subscribe to `client/+/status` but not `client/client1/status`.

You can specify the * wildcard to allow access to all actions.

`resources`

The list of resources to allow for the operations in this policy. Specify resources that correspond to the operations in this policy. For example, you might specify a list of MQTT topic resources (`mqtt:topic:`*`mqttTopic`*) in a policy that specifies the `mqtt:publish` operation.

You can specify the * wildcard to allow access to all resources. You can't use the * wildcard to match partial resource identifiers. For example, you can specify **`"resources": "*"`**, but you can't specify **`"resources": "mqtt:clientId:*"`**.

`statementDescription`

(Optional) A description for this policy statement.

`certificates`

(Optional) The certificate configuration options for this core device. This object contains the following information:

`serverCertificateValiditySeconds`

(Optional) The amount of time (in seconds) after which the local MQTT server certificate expires. You can configure this option to customize how often client devices disconnect and reconnect to the core device.

This component rotates the local MQTT server certificate 24 hours before it expires. The MQTT broker, such as the Moquette MQTT broker component, generates a new certificate and restarts. When this happens, all client devices connected to this core device are disconnected. Client devices can reconnect to the core device after a short period of time.

Default: `604800` (7 days)

Minimum value: `172800` (2 days)

Maximum value: `864000` (10 days)

`performance`

(Optional) The performance configuration options for this core device. This object contains the following information:

`maxActiveAuthTokens`

(Optional) The maximum number of active client device authorization tokens. You can increase this number to enable a greater number of client devices to connect to a single core device without reauthenticating them.

Default: `2500`

`cloudRequestQueueSize`

(Optional) The maximum number of AWS Cloud requests to queue before this component rejects requests.

Default: `100`

`maxConcurrentCloudRequests`

(Optional) The maximum number of concurrent requests to send to the AWS Cloud. You can increase this number to improve authentication performance on core devices where you connect large numbers of client devices.

Default: 1

`certificateAuthority`

(Optional) Certificate authority configuration options to replace the core device intermediate authority with your own intermediate certificate authority. This object contains the following information.

certificateUri

The location of the certificate. It can be a file system URI or a URI that points to a certificate stored in a hardware security module.

`certificateChainUri`

The location of the certificate chain for the core device CA. This should be the complete certificate chain back to your root CA. It can be a file system URI or a URI that points to a certificate chain stored in a hardware security module.

`privateKeyUri`

The location of the core device's private key. This can be a file system URI or a URI that points to a certificate private key stored in a hardware security module.

`security`

(Optional) Security configuration options for this core device. This object contains the following information.

`clientDeviceTrustDurationMinutes`

The duration in minutes that the authentication information of a client device can be trusted before it is required to reauthenticate with the core device. The default value is 1.

Example: Configuration merge update (using a restrictive policy)

The following example configuration specifies to allow client devices whose names start with `MyClientDevice` to connect and publish/subscribe on all topics.

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyDeviceGroup": {
        "selectionRule": "thingName: MyClientDevice*",
        "policyName": "MyRestrictivePolicy"
      }
    },
    "policies": {
      "MyRestrictivePolicy": {
        "AllowConnect": {
          "statementDescription": "Allow client devices to connect.",
          "operations": [
            "mqtt:connect"
          ],
          "resources": [
            "*"
          ]
        },
        "AllowPublish": {
          "statementDescription": "Allow client devices to publish on test/topic.",
```

```
          "operations": [
            "mqtt:publish"
          ],
          "resources": [
            "mqtt:topic:test/topic"
          ]
        },
        "AllowSubscribe": {
          "statementDescription": "Allow client devices to subscribe to test/topic/
response.",
          "operations": [
            "mqtt:subscribe"
          ],
          "resources": [
            "mqtt:topicfilter:test/topic/response"
          ]
        }
      }
    }
  }
}
```

Example: Configuration merge update (using a permissive policy)

The following example configuration specifies to allow all client devices to connect and publish/subscribe on all topics.

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyPermissiveDeviceGroup": {
        "selectionRule": "thingName: *",
        "policyName": "MyPermissivePolicy"
      }
    },
    "policies": {
      "MyPermissivePolicy": {
        "AllowAll": {
          "statementDescription": "Allow client devices to perform all actions.",
          "operations": [
            "*"
          ],
```

```
          "resources": [
            "*"
          ]
        }
      }
    }
  }
}
```

v2.2.x

`deviceGroups`

Device groups are groups of client devices that have permissions to connect and communicate with a core device. Use selection rules to identify groups of client devices, and define *client device authorization policies* that specify the permissions for each device group.

This object contains the following information:

`formatVersion`

The format version for this configuration object.

Choose from the following options:

- `2021-03-05`

`definitions`

The device groups for this core device. Each definition specifies a *selection rule* to evaluate if a client device is a member of the group. Each definition also specifies the permissions policy to apply to client devices that match the selection rule. If a client device is a member of multiple device groups, the device's permissions are comprised of each group's permissions policy.

This object contains the following information:

`groupNameKey`

The name of this device group. Replace *`groupNameKey`* with a name that helps you identify this device group.

This object contains the following information:

`selectionRule`

The query that specifies which client devices are members of this device group. When a client device connects, the core device evaluates this selection rule to determine if the client device is a member of this device group. If the client device is a member, the core device uses this device group's policy to authorize the client device's actions.

Each selection rule comprises at least one *selection rule clause*, which is a single expression query that can match client devices. Selection rules use the same query syntax as AWS IoT fleet indexing. For more information about selection rule syntax, see AWS IoT fleet indexing query syntax in the *AWS IoT Core Developer Guide*.

Use the `*` wildcard to match multiple client devices with one selection rule clause. You can use this wildcard at the end of the thing name to match client devices whose names start with a string that you specify. You can also use this wildcard to match all client devices.

> **Note**
>
> To select a value that contains a colon character (`:`), escape the colon with a backslash character (`\\`). In formats such as JSON, you must escape backslash characters, so you enter two backslash characters before the colon character. For example, specify `thingName:MyTeam\\\\:ClientDevice1` to select a thing whose name is `MyTeam:ClientDevice1`.

You can specify the following selector:

- `thingName` – The name of a client device's AWS IoT thing.

Example selection rule

The following selection rule matches client devices whose names are `MyClientDevice1` or `MyClientDevice2`.

```
thingName: MyClientDevice1 OR thingName: MyClientDevice2
```

Example selection rule (use wildcards)

The following selection rule matches client devices whose names start with `MyClientDevice`.

```
thingName: MyClientDevice*
```

Example selection rule (match all devices)

The following selection rule matches all client devices.

```
thingName: *
```

`policyName`

The permissions policy that applies to client devices in this device group. Specify the name of a policy that you define in the `policies` object.

`policies`

The client device authorization policies for client devices that connect to the core device. Each authorization policy specifies a set of actions and the resources where a client device can perform those actions.

This object contains the following information:

`policyNameKey`

The name of this authorization policy. Replace *`policyNameKey`* with a name that helps you identify this authorization policy. You use this policy name to define which policy applies to a device group.

This object contains the following information:

`statementNameKey`

The name of this policy statement. Replace *`statementNameKey`* with a name that helps you identify this policy statement.

This object contains the following information:

`operations`

The list of operations to allow for the resources in this policy.

You can include any of the following operations:

- `mqtt:connect` – Grants permission to connect to the core device. Client devices must have this permission to connect to a core device.

  This operation supports the following resources:

  - `mqtt:clientId:`*`deviceClientId`* – Restrict access based on the client ID that a client device uses to connect to the core device's MQTT broker. Replace *`deviceClientId`* with the client ID to use.

- `mqtt:publish` – Grants permission to publish MQTT messages to topics.

  This operation supports the following resources:

  - `mqtt:topic:`*`mqttTopic`* – Restrict access based on the MQTT topic where a client device publishes a message. Replace *`mqttTopic`* with the topic to use.

    This resource doesn't support MQTT topic wildcards.

- `mqtt:subscribe` – Grants permission to subscribe to MQTT topic filters to receive messages.

  This operation supports the following resources:

  - `mqtt:topicfilter:`*`mqttTopicFilter`* – Restrict access based on the MQTT topics where a client device can subscribe to messages. Replace *`mqttTopicFilter`* with the topic filter to use.

    This resource supports the + and # MQTT topic wildcards. For more information, see MQTT topics in the *AWS IoT Core Developer Guide*.

    The client device can subscribe to the exact topic filters that you allow. For example, if you allow the client device to subscribe to the `mqtt:topicfilter:client/+/status` resource, the client device can subscribe to `client/+/status` but not `client/client1/status`.

You can specify the * wildcard to allow access to all actions.

`resources`

The list of resources to allow for the operations in this policy. Specify resources that correspond to the operations in this policy. For example, you might specify

a list of MQTT topic resources (`mqtt:topic:`*`mqttTopic`*) in a policy that specifies the `mqtt:publish` operation.

You can specify the `*` wildcard to allow access to all resources. You can't use the `*` wildcard to match partial resource identifiers. For example, you can specify **`"resources": "*"`**, but you can't specify **`"resources": "mqtt:clientId:*"`**.

`statementDescription`

(Optional) A description for this policy statement.

`certificates`

(Optional) The certificate configuration options for this core device. This object contains the following information:

`serverCertificateValiditySeconds`

(Optional) The amount of time (in seconds) after which the local MQTT server certificate expires. You can configure this option to customize how often client devices disconnect and reconnect to the core device.

This component rotates the local MQTT server certificate 24 hours before it expires. The MQTT broker, such as the Moquette MQTT broker component, generates a new certificate and restarts. When this happens, all client devices connected to this core device are disconnected. Client devices can reconnect to the core device after a short period of time.

Default: `604800` (7 days)

Minimum value: `172800` (2 days)

Maximum value: `864000` (10 days)

`performance`

(Optional) The performance configuration options for this core device. This object contains the following information:

`maxActiveAuthTokens`

(Optional) The maximum number of active client device authorization tokens. You can increase this number to enable a greater number of client devices to connect to a single core device without reauthenticating them.

Default: 2500

`cloudRequestQueueSize`

(Optional) The maximum number of AWS Cloud requests to queue before this component rejects requests.

Default: 100

`maxConcurrentCloudRequests`

(Optional) The maximum number of concurrent requests to send to the AWS Cloud. You can increase this number to improve authentication performance on core devices where you connect large numbers of client devices.

Default: 1

Example: Configuration merge update (using a restrictive policy)

The following example configuration specifies to allow client devices whose names start with `MyClientDevice` to connect and publish/subscribe on all topics.

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyDeviceGroup": {
        "selectionRule": "thingName: MyClientDevice*",
        "policyName": "MyRestrictivePolicy"
      }
    },
    "policies": {
      "MyRestrictivePolicy": {
        "AllowConnect": {
          "statementDescription": "Allow client devices to connect.",
          "operations": [
            "mqtt:connect"
          ],
          "resources": [
            "*"
          ]
        },
        "AllowPublish": {
```

```
          "statementDescription": "Allow client devices to publish on test/topic.",
          "operations": [
            "mqtt:publish"
          ],
          "resources": [
            "mqtt:topic:test/topic"
          ]
        },
        "AllowSubscribe": {
          "statementDescription": "Allow client devices to subscribe to test/topic/
response.",
          "operations": [
            "mqtt:subscribe"
          ],
          "resources": [
            "mqtt:topicfilter:test/topic/response"
          ]
        }
      }
    }
  }
}
```

Example: Configuration merge update (using a permissive policy)

The following example configuration specifies to allow all client devices to connect and publish/subscribe on all topics.

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyPermissiveDeviceGroup": {
        "selectionRule": "thingName: *",
        "policyName": "MyPermissivePolicy"
      }
    },
    "policies": {
      "MyPermissivePolicy": {
        "AllowAll": {
          "statementDescription": "Allow client devices to perform all actions.",
          "operations": [
            "*"
```

```
          ],
          "resources": [
            "*"
          ]
        }
      }
    }
  }
}
```

v2.1.x

`deviceGroups`

Device groups are groups of client devices that have permissions to connect and communicate with a core device. Use selection rules to identify groups of client devices, and define *client device authorization policies* that specify the permissions for each device group.

This object contains the following information:

`formatVersion`

The format version for this configuration object.

Choose from the following options:

- `2021-03-05`

`definitions`

The device groups for this core device. Each definition specifies a *selection rule* to evaluate if a client device is a member of the group. Each definition also specifies the permissions policy to apply to client devices that match the selection rule. If a client device is a member of multiple device groups, the device's permissions are comprised of each group's permissions policy.

This object contains the following information:

`groupNameKey`

The name of this device group. Replace *`groupNameKey`* with a name that helps you identify this device group.

This object contains the following information:

`selectionRule`

The query that specifies which client devices are members of this device group. When a client device connects, the core device evaluates this selection rule to determine if the client device is a member of this device group. If the client device is a member, the core device uses this device group's policy to authorize the client device's actions.

Each selection rule comprises at least one *selection rule clause*, which is a single expression query that can match client devices. Selection rules use the same query syntax as AWS IoT fleet indexing. For more information about selection rule syntax, see AWS IoT fleet indexing query syntax in the *AWS IoT Core Developer Guide*.

Use the `*` wildcard to match multiple client devices with one selection rule clause. You can use this wildcard at the end of the thing name to match client devices whose names start with a string that you specify. You can also use this wildcard to match all client devices.

> **Note**
>
> To select a value that contains a colon character (`:`), escape the colon with a backslash character (`\\`). In formats such as JSON, you must escape backslash characters, so you enter two backslash characters before the colon character. For example, specify `thingName:MyTeam\\\\:ClientDevice1` to select a thing whose name is `MyTeam:ClientDevice1`.

You can specify the following selector:

- `thingName` – The name of a client device's AWS IoT thing.

Example selection rule

The following selection rule matches client devices whose names are `MyClientDevice1` or `MyClientDevice2`.

```
thingName: MyClientDevice1 OR thingName: MyClientDevice2
```

Example selection rule (use wildcards)

The following selection rule matches client devices whose names start with `MyClientDevice`.

```
thingName: MyClientDevice*
```

Example selection rule (match all devices)

The following selection rule matches all client devices.

```
thingName: *
```

`policyName`

The permissions policy that applies to client devices in this device group. Specify the name of a policy that you define in the `policies` object.

`policies`

The client device authorization policies for client devices that connect to the core device. Each authorization policy specifies a set of actions and the resources where a client device can perform those actions.

This object contains the following information:

`policyNameKey`

The name of this authorization policy. Replace *`policyNameKey`* with a name that helps you identify this authorization policy. You use this policy name to define which policy applies to a device group.

This object contains the following information:

`statementNameKey`

The name of this policy statement. Replace *`statementNameKey`* with a name that helps you identify this policy statement.

This object contains the following information:

`operations`

The list of operations to allow for the resources in this policy.

You can include any of the following operations:

- `mqtt:connect` – Grants permission to connect to the core device. Client devices must have this permission to connect to a core device.

  This operation supports the following resources:

  - `mqtt:clientId:`*`deviceClientId`* – Restrict access based on the client ID that a client device uses to connect to the core device's MQTT broker. Replace *`deviceClientId`* with the client ID to use.

- `mqtt:publish` – Grants permission to publish MQTT messages to topics.

  This operation supports the following resources:

  - `mqtt:topic:`*`mqttTopic`* – Restrict access based on the MQTT topic where a client device publishes a message. Replace *`mqttTopic`* with the topic to use.

    This resource doesn't support MQTT topic wildcards.

- `mqtt:subscribe` – Grants permission to subscribe to MQTT topic filters to receive messages.

  This operation supports the following resources:

  - `mqtt:topicfilter:`*`mqttTopicFilter`* – Restrict access based on the MQTT topics where a client device can subscribe to messages. Replace *`mqttTopicFilter`* with the topic filter to use.

    This resource supports the + and # MQTT topic wildcards. For more information, see MQTT topics in the *AWS IoT Core Developer Guide*.

    The client device can subscribe to the exact topic filters that you allow. For example, if you allow the client device to subscribe to the `mqtt:topicfilter:client/+/status` resource, the client device can subscribe to `client/+/status` but not `client/client1/status`.

You can specify the * wildcard to allow access to all actions.

`resources`

The list of resources to allow for the operations in this policy. Specify resources that correspond to the operations in this policy. For example, you might specify

a list of MQTT topic resources (`mqtt:topic:`*`mqttTopic`*) in a policy that specifies the `mqtt:publish` operation.

You can specify the `*` wildcard to allow access to all resources. You can't use the `*` wildcard to match partial resource identifiers. For example, you can specify **`"resources": "*"`**, but you can't specify **`"resources": "mqtt:clientId:*"`**.

`statementDescription`

(Optional) A description for this policy statement.

`certificates`

(Optional) The certificate configuration options for this core device. This object contains the following information:

`serverCertificateValiditySeconds`

(Optional) The amount of time (in seconds) after which the local MQTT server certificate expires. You can configure this option to customize how often client devices disconnect and reconnect to the core device.

This component rotates the local MQTT server certificate 24 hours before it expires. The MQTT broker, such as the Moquette MQTT broker component, generates a new certificate and restarts. When this happens, all client devices connected to this core device are disconnected. Client devices can reconnect to the core device after a short period of time.

Default: `604800` (7 days)

Minimum value: `172800` (2 days)

Maximum value: `864000` (10 days)

Example: Configuration merge update (using a restrictive policy)

The following example configuration specifies to allow client devices whose names start with `MyClientDevice` to connect and publish/subscribe on all topics.

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
```

```
    "definitions": {
      "MyDeviceGroup": {
        "selectionRule": "thingName: MyClientDevice*",
        "policyName": "MyRestrictivePolicy"
      }
    },
    "policies": {
      "MyRestrictivePolicy": {
        "AllowConnect": {
          "statementDescription": "Allow client devices to connect.",
          "operations": [
            "mqtt:connect"
          ],
          "resources": [
            "*"
          ]
        },
        "AllowPublish": {
          "statementDescription": "Allow client devices to publish on test/topic.",
          "operations": [
            "mqtt:publish"
          ],
          "resources": [
            "mqtt:topic:test/topic"
          ]
        },
        "AllowSubscribe": {
          "statementDescription": "Allow client devices to subscribe to test/topic/
response.",
          "operations": [
            "mqtt:subscribe"
          ],
          "resources": [
            "mqtt:topicfilter:test/topic/response"
          ]
        }
      }
    }
  }
}
```

Example: Configuration merge update (using a permissive policy)

The following example configuration specifies to allow all client devices to connect and publish/subscribe on all topics.

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyPermissiveDeviceGroup": {
        "selectionRule": "thingName: *",
        "policyName": "MyPermissivePolicy"
      }
    },
    "policies": {
      "MyPermissivePolicy": {
        "AllowAll": {
          "statementDescription": "Allow client devices to perform all actions.",
          "operations": [
            "*"
          ],
          "resources": [
            "*"
          ]
        }
      }
    }
  }
}
```

v2.0.x

`deviceGroups`

Device groups are groups of client devices that have permissions to connect and communicate with a core device. Use selection rules to identify groups of client devices, and define *client device authorization policies* that specify the permissions for each device group.

This object contains the following information:

`formatVersion`

The format version for this configuration object.

Choose from the following options:

- `2021-03-05`

`definitions`

The device groups for this core device. Each definition specifies a *selection rule* to evaluate if a client device is a member of the group. Each definition also specifies the permissions policy to apply to client devices that match the selection rule. If a client device is a member of multiple device groups, the device's permissions are comprised of each group's permissions policy.

This object contains the following information:

`groupNameKey`

The name of this device group. Replace *`groupNameKey`* with a name that helps you identify this device group.

This object contains the following information:

`selectionRule`

The query that specifies which client devices are members of this device group. When a client device connects, the core device evaluates this selection rule to determine if the client device is a member of this device group. If the client device is a member, the core device uses this device group's policy to authorize the client device's actions.

Each selection rule comprises at least one *selection rule clause*, which is a single expression query that can match client devices. Selection rules use the same query syntax as AWS IoT fleet indexing. For more information about selection rule syntax, see AWS IoT fleet indexing query syntax in the *AWS IoT Core Developer Guide*.

Use the * wildcard to match multiple client devices with one selection rule clause. You can use this wildcard at the end of the thing name to match client devices whose names start with a string that you specify. You can also use this wildcard to match all client devices.

> **Note**
>
> To select a value that contains a colon character (`:`), escape the colon with a backslash character (`\\`). In formats such as JSON, you must escape backslash characters, so you enter two backslash characters before the colon character. For example, specify `thingName:MyTeam\\\\:ClientDevice1` to select a thing whose name is `MyTeam:ClientDevice1`.

You can specify the following selector:

- `thingName` – The name of a client device's AWS IoT thing.

Example selection rule

The following selection rule matches client devices whose names are `MyClientDevice1` or `MyClientDevice2`.

```
thingName: MyClientDevice1 OR thingName: MyClientDevice2
```

Example selection rule (use wildcards)

The following selection rule matches client devices whose names start with `MyClientDevice`.

```
thingName: MyClientDevice*
```

Example selection rule (match all devices)

The following selection rule matches all client devices.

```
thingName: *
```

`policyName`

The permissions policy that applies to client devices in this device group. Specify the name of a policy that you define in the `policies` object.

`policies`

The client device authorization policies for client devices that connect to the core device. Each authorization policy specifies a set of actions and the resources where a client device can perform those actions.

This object contains the following information:

`policyNameKey`

The name of this authorization policy. Replace *`policyNameKey`* with a name that helps you identify this authorization policy. You use this policy name to define which policy applies to a device group.

This object contains the following information:

`statementNameKey`

The name of this policy statement. Replace *`statementNameKey`* with a name that helps you identify this policy statement.

This object contains the following information:

`operations`

The list of operations to allow for the resources in this policy.

You can include any of the following operations:

- `mqtt:connect` – Grants permission to connect to the core device. Client devices must have this permission to connect to a core device.

  This operation supports the following resources:

  - `mqtt:clientId:`*`deviceClientId`* – Restrict access based on the client ID that a client device uses to connect to the core device's MQTT broker. Replace *`deviceClientId`* with the client ID to use.
- `mqtt:publish` – Grants permission to publish MQTT messages to topics.

  This operation supports the following resources:

  - `mqtt:topic:`*`mqttTopic`* – Restrict access based on the MQTT topic where a client device publishes a message. Replace *`mqttTopic`* with the topic to use.

    This resource doesn't support MQTT topic wildcards.

- `mqtt:subscribe` – Grants permission to subscribe to MQTT topic filters to receive messages.

  This operation supports the following resources:

  - `mqtt:topicfilter:`*`mqttTopicFilter`* – Restrict access based on the MQTT topics where a client device can subscribe to messages. Replace *`mqttTopicFilter`* with the topic filter to use.

    This resource supports the + and # MQTT topic wildcards. For more information, see MQTT topics in the *AWS IoT Core Developer Guide*.

    The client device can subscribe to the exact topic filters that you allow. For example, if you allow the client device to subscribe to the `mqtt:topicfilter:client/+/status` resource, the client device can subscribe to `client/+/status` but not `client/client1/status`.

  You can specify the * wildcard to allow access to all actions.

`resources`

The list of resources to allow for the operations in this policy. Specify resources that correspond to the operations in this policy. For example, you might specify a list of MQTT topic resources (`mqtt:topic:`*`mqttTopic`*) in a policy that specifies the `mqtt:publish` operation.

You can specify the * wildcard to allow access to all resources. You can't use the * wildcard to match partial resource identifiers. For example, you can specify **`"resources": "*"`**, but you can't specify **`"resources": "mqtt:clientId:*"`**.

`statementDescription`

(Optional) A description for this policy statement.

Example: Configuration merge update (using a restrictive policy)

The following example configuration specifies to allow client devices whose names start with `MyClientDevice` to connect and publish/subscribe on all topics.

```
{
  "deviceGroups": {
```

```
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyDeviceGroup": {
        "selectionRule": "thingName: MyClientDevice*",
        "policyName": "MyRestrictivePolicy"
      }
    },
    "policies": {
      "MyRestrictivePolicy": {
        "AllowConnect": {
          "statementDescription": "Allow client devices to connect.",
          "operations": [
            "mqtt:connect"
          ],
          "resources": [
            "*"
          ]
        },
        "AllowPublish": {
          "statementDescription": "Allow client devices to publish on test/topic.",
          "operations": [
            "mqtt:publish"
          ],
          "resources": [
            "mqtt:topic:test/topic"
          ]
        },
        "AllowSubscribe": {
          "statementDescription": "Allow client devices to subscribe to test/topic/
response.",
          "operations": [
            "mqtt:subscribe"
          ],
          "resources": [
            "mqtt:topicfilter:test/topic/response"
          ]
        }
      }
    }
  }
}
```

Example: Configuration merge update (using a permissive policy)

The following example configuration specifies to allow all client devices to connect and publish/subscribe on all topics.

```
{
  "deviceGroups": {
    "formatVersion": "2021-03-05",
    "definitions": {
      "MyPermissiveDeviceGroup": {
        "selectionRule": "thingName: *",
        "policyName": "MyPermissivePolicy"
      }
    },
    "policies": {
      "MyPermissivePolicy": {
        "AllowAll": {
          "statementDescription": "Allow client devices to perform all actions.",
          "operations": [
            "*"
          ],
          "resources": [
            "*"
          ]
        }
      }
    }
  }
}
```

Local log file

This component uses the same log file as the Greengrass nucleus component.

Linux

```
/greengrass/v2/logs/greengrass.log
```

Windows

```
C:\greengrass\v2\logs\greengrass.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
  ```

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.5.7	Updates the component version for the Greengrass nucleus version 2.18.0 release.
2.5.6	Updates the component version for the Greengrass nucleus version 2.17.0 release.
2.5.5	Version updated for Greengrass nucleus version 2.16.0 release.
2.5.4	Version updated for Greengrass nucleus version 2.15.0 release.
2.5.3	Bug fixes and improvements • Fixes an issue where client devices are unable to connect to the core device due to out of date client certificates.
2.5.2	Version updated for Greengrass nucleus version 2.14.0 release.
2.5.1	Bug fixes and improvements • Supports FIPS endpoint.

Version	Changes
2.5.0	New features • Allows `${iot:Connection.Thing.ThingName}` variable substitution for policy resources. • Allows policy resources with wildcards such as `mqtt:topic:my*` . Bug fixes and improvements • Fixes an issue where the client device auth takes more than 10 seconds to verify a client device's identity. • Additional minor fixes and improvements.
2.4.5	New features Adds support for wildcard prefixes for selecting thing names with the `selectionRule` parameter. Bug fixes and improvements Fixes an issue where certificates aren't updated with new connectivity information in certain cases.
2.4.4	Version updated for Greengrass nucleus version 2.12.0 release.
2.4.3	Version updated for Greengrass nucleus version 2.11.0 release.
2.4.2	New features Adds a new `startupTimeoutSeconds` configuration option.
2.4.1	Version updated for Greengrass nucleus version 2.10.0 release.
2.4.0	New features • Adds support for client device auth to emit operational metrics that will be published by the telemetry agent. Bug fixes and improvements • Fixes an issue where the client device auth takes more than 10 seconds to verify a client device's identity. • Additional minor fixes and improvements.
2.3.2	Bug fixes and improvements • Adds support for caching hostname information so that the component correctly generates certificate subjects when restarted when offline.

Version	Changes
2.3.1	Bug fixes and improvements • Fixes a memory leak.
2.3.0	**Warning** This version is no longer available. The improvements in this version are available in later versions of this component. New features • Adds support for offline authentication of client devices so that they can continue to connect to the core device when the core device isn't connected to the Internet. • Adds support for customer-provided certificate authority that the core device uses as the root certificate to generate MQTT broker certificates.
2.2.3	Version updated for Greengrass nucleus version 2.8.0 release.
2.2.2	Bug fixes and improvements • Fixes an issue where the local MQTT server certificate rotates more often than intended in certain scenarios.
2.2.1	Version updated for Greengrass nucleus version 2.7.0 release.
2.2.0	New features • Adds support for custom components to call interprocess communication (IPC) operations to authenticate and authorize client devices. You can use these operations in a custom MQTT broker component, for example. For more information, see IPC: Authenticate and authorize client devices. • Adds the `maxActiveAuthTokens` , `cloudQueueSize` , and `threadPoolSize` options that you can configure to tune how this component performs.

Version	Changes
2.1.0	New features • Adds the `serverCertificateValiditySeconds` option that you can configure to customize when the MQTT broker server certificate expires. You can configure the server certificate to expire after 2 to 10 days. Bug fixes and improvements • Fixes issues with how this component handles configuration reset updates. • Fixes an issue where the local MQTT server certificate rotates more often than intended in certain scenarios. To apply this fix, you must also use v2.1.0 or later of the Moquette MQTT broker component. • Improves messages that this component logs when it rotates certificates. • Version updated for Greengrass nucleus version 2.6.0 release.
2.0.4	Version updated for Greengrass nucleus version 2.5.0 release.
2.0.3	Bug fixes and improvements • Credentials now refresh if you rotate the core device's private key. • Updates to make log messages more clear.
2.0.2	Version updated for Greengrass nucleus version 2.4.0 release.
2.0.1	Version updated for Greengrass nucleus version 2.3.0 release.
2.0.0	Initial version.

CloudWatch metrics

The Amazon CloudWatch metrics component (`aws.greengrass.Cloudwatch`) publishes custom metrics from Greengrass core devices to Amazon CloudWatch. The component enables components to publish CloudWatch metrics, which you can use to monitor and analyze the

Greengrass core device's environment. For more information, see Using Amazon CloudWatch metrics in the *Amazon CloudWatch User Guide*.

To publish a CloudWatch metric with this component, publish a message to a topic where this component subscribes. By default, this component subscribes to the `cloudwatch/metric/put` local publish/subscribe topic. You can specify other topics, including AWS IoT Core MQTT topics, when you deploy this component.

This component batches metrics that are in the same namespace and publishes them to CloudWatch at regular intervals.

> **Note**
>
> This component provides similar functionality to the CloudWatch metrics connector in AWS IoT Greengrass V1. For more information, see CloudWatch metrics connector in the *AWS IoT Greengrass V1 Developer Guide*.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Input data
- Output data
- Licenses
- Local log file
- Changelog
- See also

Versions

This component has the following versions:

- 4.0.x
- 3.2.x
- 3.1.x
- 3.0.x
- 2.1.x
- 2.0.x

For information about changes in each version of the component, see the changelog.

Type

v4.x - v3.x

This component is a generic component (`aws.greengrass.generic`). The Greengrass nucleus runs the component's lifecycle scripts.

v2.x

This component is a Lambda component (`aws.greengrass.lambda`). The Greengrass nucleus runs this component's Lambda function using the Lambda launcher component.

For more information, see Component types.

Operating system

v4.x - v3.x

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

v2.x

This component can be installed on Linux core devices only.

Requirements

This component has the following requirements:

v4.x - v3.x

- Python version 3.7 installed on the core device and added to the PATH environment variable.
- The following Python libraries, including any dependencies, must be installed and available to the user running the component:
  - AWS IoT Device SDK v2 for Python
  - AWS SDK for Python (Boto3)

  > **Note**
  >
  > You can set the `UseInstaller` configuration to `true` to install these libraries automatically into the provided Python environment.

- The Greengrass device role must allow the `cloudwatch:PutMetricData` action, as shown in the following example IAM policy.

  JSON

  ```
  {
    "Version":"2012-10-17",
    "Statement": [
      {
        "Action": [
          "cloudwatch:PutMetricData"
        ],
        "Effect": "Allow",
        "Resource": "*"
      }
    ]
  }
  ```

  For more information, see Amazon CloudWatch permissions reference in the *Amazon CloudWatch User Guide*.

2.x

- Your core device must meet the requirements to run Lambda functions. If you want the core device to run containerized Lambda functions, the device must meet the requirements to do so. For more information, see Lambda function requirements.
- Python version 3.7 installed on the core device and added to the PATH environment variable.
- The Greengrass device role must allow the `cloudwatch:PutMetricData` action, as shown in the following example IAM policy.

  JSON

  ```
  {
    "Version":"2012-10-17",
    "Statement": [
      {
        "Action": [
          "cloudwatch:PutMetricData"
        ],
        "Effect": "Allow",
        "Resource": "*"
      }
    ]
  }
  ```

  For more information, see Amazon CloudWatch permissions reference in the *Amazon CloudWatch User Guide*.
- To receive output data from this component, you must merge the following configuration update for the legacy subscription router component (`aws.greengrass.LegacySubscriptionRouter`) when you deploy this component. This configuration specifies the topic where this component publishes responses.

  Legacy subscription router v2.1.x

  ```
  {
    "subscriptions": {
      "aws-greengrass-cloudwatch": {
        "id": "aws-greengrass-cloudwatch",
        "source": "component:aws.greengrass.Cloudwatch",
        "subject": "cloudwatch/metric/put/status",
        "target": "cloud"
      }
  ```

```
  }
}
```

Legacy subscription router v2.0.x

```
{
  "subscriptions": {
    "aws-greengrass-cloudwatch": {
      "id": "aws-greengrass-cloudwatch",
      "source": "arn:aws:lambda:region:aws:function:aws-greengrass-
cloudwatch:version",
      "subject": "cloudwatch/metric/put/status",
      "target": "cloud"
    }
  }
}
```

- Replace *region* with the AWS Region that you use.
- Replace *version* with the version of the Lambda function that this component runs. To find the Lambda function version, you must view the recipe for the version of this component that you want to deploy. Open this component's details page in the AWS IoT Greengrass console, and look for the **Lambda function** key-value pair. This key-value pair contains the name and version of the Lambda function.

> **Important**
>
> You must update the Lambda function version on the legacy subscription router every time you deploy this component. This ensures that you use the correct Lambda function version for the component version that you deploy.

For more information, see Create deployments.

Endpoints and ports

This component must be able to perform outbound requests to the following endpoints and ports, in addition to endpoints and ports required for basic operation. For more information, see Allow device traffic through a proxy or firewall.

Endpoint	Port	Required	Description
`monitoring. region.amazonaws.com`	443	Yes	Upload CloudWatch metrics.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

4.0.0 - 3.0.0

The following table lists the dependencies for versions 4.0.0 to 3.0.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <3.0.0	Soft
Token exchange service	>=0.0.0	Hard

2.1.4 - 2.1.9

The following table lists the dependencies for versions 2.1.4 to 2.1.9 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <3.0.0	Hard
Lambda launcher	^2.0.0	Hard

Dependency	Compatible versions	Dependency type
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.1.4 - 2.1.8

The following table lists the dependencies for version 2.1.4 and 2.1.8 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <3.0.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.1.2 - 2.1.3

The following table lists the dependencies for version 2.1.2 and 2.1.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.8.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.1.1

The following table lists the dependencies for version 2.1.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.7.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.0.8 - 2.1.0

The following table lists the dependencies for versions 2.0.8 to 2.1.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.6.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.0.7

The following table lists the dependencies for version 2.0.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.5.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.0.6

The following table lists the dependencies for version 2.0.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.4.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.0.5

The following table lists the dependencies for version 2.0.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.3.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.0.4

The following table lists the dependencies for version 2.0.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.2.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft

Dependency	Compatible versions	Dependency type
Token exchange service	^2.0.0	Hard

2.0.3

The following table lists the dependencies for version 2.0.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.3 <2.1.0	Hard
Lambda launcher	>=1.0.0	Hard
Lambda runtimes	>=1.0.0	Soft
Token exchange service	>=1.0.0	Hard

For more information about component dependencies, see the component recipe reference.

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

v4.x

`PublishInterval`

(Optional) The maximum number of seconds to wait before the component publishes batched metrics for a given namespace. To configure the component to publish metrics as it receives them, which means without batching, specify `0`.

The component publishes to CloudWatch after it receives 20 metrics in the same namespace or after the interval that you specify.

> **Note**
>
> The component doesn't specify the order in which events publish.

This value can be a maximum of 900 seconds.

Default: 10 seconds

`MaxMetricsToRetain`

(Optional) The maximum number of metrics across all namespaces to save in memory before the component replaces them with newer metrics.

This limit applies when the core device doesn't have a connection to the internet, so the component buffers the metrics to publish later. When the buffer is full, the component replaces the oldest metrics with newer ones. Metrics in a given namespace replace only metrics in the same namespace.

> **Note**
>
> If the host process for the component is interrupted, the component doesn't save metrics. This can happen during a deployment or when the core device restarts, for example.

This value must be at least 2,000 metrics.

Default: 5,000 metrics

`InputTopic`

(Optional) The topic to which the component subscribes to receive messages. If you specify `true` for `PubSubToIoTCore`, you can use MQTT wildcards (+ and #) in this topic.

Default: `cloudwatch/metric/put`

`OutputTopic`

(Optional) The topic to which the component publishes status responses.

Default: `cloudwatch/metric/put/status`

`PubSubToIoTCore`

(Optional) String value that defines whether to publish and subscribe to AWS IoT Core MQTT topics. Supported values are `true` and `false`.

Default: `false`

`LogLevel`

(Optional) The logging level for the component. Choose from the following log levels, listed here in level order:

- `DEBUG`
- `INFO`
- `WARNING`
- `ERROR`
- `CRITICAL`

Default: `INFO`

`UseInstaller`

(Optional) Boolean value that defines whether to use the installer script to install this component's Python dependencies into the provided Python environment.

Set this value to `true` to automatically install this component's Python dependencies. When set to `false`, you must install the following libraries, including any dependencies, and make them available to the Python environment provided to the user running the component.

- AWS IoT Device SDK v2 for Python
- AWS SDK for Python (Boto3)

Default: `false`

`PublishRegion`

(Optional) The AWS Region to which to publish CloudWatch metrics. This value overrides the default Region for the core device. This parameter is required only for cross-Region metrics.

`accessControl`

(Optional) The object that contains the authorization policy that allows the component to publish and subscribe to the specified topics. If you specify custom values for `InputTopic` and `OutputTopic`, you must update the resource values in this object.

Default:

```
{
  "aws.greengrass.ipc.pubsub": {
    "aws.greengrass.Cloudwatch:pubsub:1": {
      "policyDescription": "Allows access to subscribe to input topics.",
```

```
      "operations": [
        "aws.greengrass#SubscribeToTopic"
      ],
      "resources": [
        "cloudwatch/metric/put"
      ]
    },
    "aws.greengrass.Cloudwatch:pubsub:2": {
      "policyDescription": "Allows access to publish to output topics.",
      "operations": [
        "aws.greengrass#PublishToTopic"
      ],
      "resources": [
        "cloudwatch/metric/put/status"
      ]
    }
  },
  "aws.greengrass.ipc.mqttproxy": {
    "aws.greengrass.Cloudwatch:mqttproxy:1": {
      "policyDescription": "Allows access to subscribe to input topics.",
      "operations": [
        "aws.greengrass#SubscribeToIoTCore"
      ],
      "resources": [
        "cloudwatch/metric/put"
      ]
    },
    "aws.greengrass.Cloudwatch:mqttproxy:2": {
      "policyDescription": "Allows access to publish to output topics.",
      "operations": [
        "aws.greengrass#PublishToIoTCore"
      ],
      "resources": [
        "cloudwatch/metric/put/status"
      ]
    }
  }
}
```

Example: Configuration merge update

```
{
```

```
  "PublishInterval": 0,
  "PubSubToIoTCore": true
}
```

v3.x

`PublishInterval`

(Optional) The maximum number of seconds to wait before the component publishes batched metrics for a given namespace. To configure the component to publish metrics as it receives them, which means without batching, specify `0`.

The component publishes to CloudWatch after it receives 20 metrics in the same namespace or after the interval that you specify.

> **Note**
>
> The component doesn't specify the order in which events publish.

This value can be a maximum of 900 seconds.

Default: 10 seconds

`MaxMetricsToRetain`

(Optional) The maximum number of metrics across all namespaces to save in memory before the component replaces them with newer metrics.

This limit applies when the core device doesn't have a connection to the internet, so the component buffers the metrics to publish later. When the buffer is full, the component replaces the oldest metrics with newer ones. Metrics in a given namespace replace only metrics in the same namespace.

> **Note**
>
> If the host process for the component is interrupted, the component doesn't save metrics. This can happen during a deployment or when the core device restarts, for example.

This value must be at least 2,000 metrics.

Default: 5,000 metrics

`InputTopic`

(Optional) The topic to which the component subscribes to receive messages. If you specify `true` for `PubSubToIoTCore`, you can use MQTT wildcards (+ and #) in this topic.

Default: `cloudwatch/metric/put`

`OutputTopic`

(Optional) The topic to which the component publishes status responses.

Default: `cloudwatch/metric/put/status`

`PubSubToIoTCore`

(Optional) String value that defines whether to publish and subscribe to AWS IoT Core MQTT topics. Supported values are `true` and `false`.

Default: `false`

`LogLevel`

(Optional) The logging level for the component. Choose from the following log levels, listed here in level order:

- `DEBUG`
- `INFO`
- `WARNING`
- `ERROR`
- `CRITICAL`

Default: `INFO`

`UseInstaller`

(Optional) Boolean value that defines whether to use the installer script to install this component's Python dependencies into the provided Python environment.

Set this value to `false` to use a custom script or manually install dependencies. When set to `false`, you must install the following libraries, including any dependencies, and make them available to the Python environment provided to the user running the component.

- AWS IoT Device SDK v2 for Python
- AWS SDK for Python (Boto3)

Default: `true`

`PublishRegion`

(Optional) The AWS Region to which to publish CloudWatch metrics. This value overrides the default Region for the core device. This parameter is required only for cross-Region metrics.

`accessControl`

(Optional) The object that contains the authorization policy that allows the component to publish and subscribe to the specified topics. If you specify custom values for `InputTopic` and `OutputTopic`, you must update the resource values in this object.

Default:

```
{
  "aws.greengrass.ipc.pubsub": {
    "aws.greengrass.Cloudwatch:pubsub:1": {
      "policyDescription": "Allows access to subscribe to input topics.",
      "operations": [
        "aws.greengrass#SubscribeToTopic"
      ],
      "resources": [
        "cloudwatch/metric/put"
      ]
    },
    "aws.greengrass.Cloudwatch:pubsub:2": {
      "policyDescription": "Allows access to publish to output topics.",
      "operations": [
        "aws.greengrass#PublishToTopic"
      ],
      "resources": [
        "cloudwatch/metric/put/status"
      ]
    }
  },
  "aws.greengrass.ipc.mqttproxy": {
    "aws.greengrass.Cloudwatch:mqttproxy:1": {
      "policyDescription": "Allows access to subscribe to input topics.",
      "operations": [
        "aws.greengrass#SubscribeToIoTCore"
      ],
      "resources": [
        "cloudwatch/metric/put"
```

```
      ]
    },
    "aws.greengrass.Cloudwatch:mqttproxy:2": {
      "policyDescription": "Allows access to publish to output topics.",
      "operations": [
        "aws.greengrass#PublishToIoTCore"
      ],
      "resources": [
        "cloudwatch/metric/put/status"
      ]
    }
  }
}
```

Example: Configuration merge update

```
{
  "PublishInterval": 0,
  "PubSubToIoTCore": true
}
```

v2.x

> **Note**
>
> This component's default configuration includes Lambda function parameters. We recommend that you edit only the following parameters to configure this component on your devices.

`lambdaParams`

An object that contains the parameters for this component's Lambda function. This object contains the following information:

`EnvironmentVariables`

An object that contains the Lambda function's parameters. This object contains the following information:

`PUBLISH_INTERVAL`

(Optional) The maximum number of seconds to wait before the component publishes batched metrics for a given namespace. To configure the component to publish metrics as it receives them, which means without batching, specify `0`.

The component publishes to CloudWatch after it receives 20 metrics in the same namespace or after the interval that you specify.

> **Note**
>
> The component doesn't guarantee the order in which events publish.

This value can be at most 900 seconds.

Default: 10 seconds

`MAX_METRICS_TO_RETAIN`

(Optional) The maximum number of metrics across all namespaces to save in memory before the component replaces them with newer metrics.

This limit applies when the core device doesn't have a connection to the internet, so the component buffers the metrics to publish later. When the buffer is full, the component replaces the oldest metrics with newer ones. Metrics in a given namespace replace only metrics in the same namespace.

> **Note**
>
> If the host process for the component is interrupted, the component doesn't save metrics. This can happen during a deployment or when the core device restarts, for example.

This value must be at least 2,000 metrics.

Default: 5,000 metrics

`PUBLISH_REGION`

(Optional) The AWS Region to which to publish CloudWatch metrics. This value overrides the default Region for the core device. This parameter is required only for cross-Region metrics.

`containerMode`

(Optional) The containerization mode for this component. Choose from the following options:

- `NoContainer` – The component doesn't run in an isolated runtime environment.
- `GreengrassContainer` – The component runs in an isolated runtime environment inside the AWS IoT Greengrass container.

Default: `GreengrassContainer`

`containerParams`

(Optional) An object that contains the container parameters for this component. The component uses these parameters if you specify `GreengrassContainer` for `containerMode`.

This object contains the following information:

`memorySize`

(Optional) The amount of memory (in kilobytes) to allocate to the component.

Defaults to 64 MB (65,535 KB).

`pubsubTopics`

(Optional) An object that contains the topics where the component subscribes to receive messages. You can specify each topic and whether the component subscribes to MQTT topics from AWS IoT Core or local publish/subscribe topics.

This object contains the following information:

`0` – This is an array index as a string.

An object that contains the following information:

`type`

(Optional) The type of publish/subscribe messaging that this component uses to subscribe to messages. Choose from the following options:

- PUB_SUB – Subscribe to local publish/subscribe messages. If you choose this option, the topic can't contain MQTT wildcards. For more information about how to send messages from custom component when you specify this option, see Publish/subscribe local messages.
- `IOT_CORE` – Subscribe to AWS IoT Core MQTT messages. If you choose this option, the topic can contain MQTT wildcards. For more information about how to send messages from custom components when you specify this option, see Publish/subscribe AWS IoT Core MQTT messages.

Default: PUB_SUB

`topic`

(Optional) The topic to which the component subscribes to receive messages. If you specify `IotCore` for `type`, you can use MQTT wildcards (+ and #) in this topic.

Example: Configuration merge update (container mode)

```
{
  "containerMode": "GreengrassContainer"
}
```

Example: Configuration merge update (no container mode)

```
{
  "containerMode": "NoContainer"
}
```

Input data

This component accepts metrics on the following topic and publishes the metrics to CloudWatch. By default, this component subscribes to local publish/subscribe messages. For more information about how to publish messages to this component from your custom components, see Publish/subscribe local messages.

Beginning with component version v3.0.0, you can optionally configure this component to subscribe to an MQTT topic by setting the `PubSubToIoTCore` configuration parameter to `true`. For more information about publishing messages to an MQTT topic in your custom components, see Publish/subscribe AWS IoT Core MQTT messages.

Default topic: `cloudwatch/metric/put`

The message accepts the following properties. Input messages must be in JSON format.

`request`

The metric in this message.

The request object contains the metric data to publish to CloudWatch. The metric values must meet the specifications of the `PutMetricData` operation.

Type: `object` that contains the following information:

`namespace`

The user-defined namespace for the metric data in this request. CloudWatch uses namespaces as containers for metric data points.

> **Note**
>
> You can't specify a namespace that begins with the reserved string `AWS/`.

Type: `string`

Valid pattern: `[^:].*`

`metricData`

The data for the metric.

Type: `object` that contains the following information:

`metricName`

The name of the metric.

Type: `string`

`value`

The value for the metric.

> **Note**
>
> CloudWatch rejects values that are too small or too large. The value must be between `8.515920e-109` and `1.174271e+108` (Base 10) or `2e-360` and `2e360` (Base 2). CloudWatch doesn't support special values such as `NaN`, `+Infinity`, and `-Infinity`.

Type: `double`

`dimensions`

(Optional) The dimensions for the metric. Dimensions provide additional information about the metric and its data. A metric can define up to 10 dimensions.

This component automatically includes a dimension named `coreName`, where the value is the name of the core device.

Type: `array` of objects that each contain the following information:

`name`

(Optional) The dimension name.

Type: `string`

`value`

(Optional) The dimension value.

Type: `string`

`timestamp`

(Optional) The time at which the metric data was received, expressed in seconds in Unix epoch time.

Defaults to the time at which the component receives the message.

Type: `double`

> **Note**
>
> If you use between versions 2.0.3 and 2.0.7 of this component, we recommend that you retrieve the timestamp separately for each metric when you send

multiple metrics from a single source. Don't use a variable to store the timestamp.

`unit`

(Optional) The unit of the metric.

Type: `string`

Valid values: `Seconds`, `Microseconds`, `Milliseconds`, `Bytes`, `Kilobytes`, `Megabytes`, `Gigabytes`, `Terabytes`, `Bits`, `Kilobits`, `Megabits`, `Gigabits`, `Terabits`, `Percent`, `Count`, `Bytes/Second`, `Kilobytes/Second`, `Megabytes/Second`, `Gigabytes/Second`, `Terabytes/Second`, `Bits/Second`, `Kilobits/Second`, `Megabits/Second`, `Gigabits/Second`, `Terabits/Second`, `Count/Second`, `None`

Defaults to `None`.

Note

All quotas that apply to the CloudWatch `PutMetricData` API apply to metrics that you publish with this component. The following quotas are especially important:

- 40 KB limit on the API payload
- 20 metrics per API request
- 150 transactions per second (TPS) for the `PutMetricData` API

For more information, see CloudWatch service quotas in the *CloudWatch User Guide*.

Example input

```
{
  "request": {
    "namespace": "Greengrass",
    "metricData": {
      "metricName": "latency",
```

```
      "dimensions": [
        {
          "name": "hostname",
          "value": "test_hostname"
        }
      ],
      "timestamp": 1539027324,
      "value": 123.0,
      "unit": "Seconds"
    }
  }
}
```

Output data

This component publishes responses as output data on the following local publish/subscribe topic by default. For more information about how to subscribe to messages on this topic in your custom components, see Publish/subscribe local messages.

You can optionally configure this component to publish to an MQTT topic by setting the `PubSubToIoTCore` configuration parameter to `true`. For more information about subscribing to messages on an MQTT topic in your custom components, see Publish/subscribe AWS IoT Core MQTT messages.

> **Note**
>
> Component versions 2.0.x publish responses as output data on an MQTT topic by default. You must specify the topic as the `subject` in the configuration for the legacy subscription router component.

Default topic: `cloudwatch/metric/put/status`

Example output: Success

The response includes the namespace of the metric data and the `RequestId` field from the CloudWatch response.

```
{
  "response": {
```

```
    "cloudwatch_rid": "70573243-d723-11e8-b095-75ff2EXAMPLE",
    "namespace": "Greengrass",
    "status": "success"
  }
}
```

Example output: Failure

```
{
  "response" : {
    "namespace": "Greengrass",
    "error": "InvalidInputException",
    "error_message": "cw metric is invalid",
    "status": "fail"
  }
}
```

> **Note**
>
> If the component detects an error that can be retried, such as a connection error, it retries the publish in the next batch.

Licenses

This component includes the following third-party software/licensing:

- AWS SDK for Python (Boto3)/Apache License 2.0
- botocore/Apache License 2.0
- dateutil/PSF License
- docutils/BSD License, GNU General Public License (GPL), Python Software Foundation License, Public Domain
- jmespath/MIT License
- s3transfer/Apache License 2.0
- urllib3/MIT License

This component is released under the Greengrass Core Software License Agreement.

Local log file

This component uses the following log file.

Linux

```
/greengrass/v2/logs/aws.greengrass.Cloudwatch.log
```

Windows

```
C:\greengrass\v2\logs\aws.greengrass.Cloudwatch.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/aws.greengrass.Cloudwatch.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\aws.greengrass.Cloudwatch.log -Tail 10 -Wait
  ```

Changelog

The following table describes the changes in each version of the component.

v4.x - v3.x

Version	Changes
4.0.0	New features • The `UseInstaller` configuration now defaults to `false`.

Version	Changes
3.2.0	New features • Add recipe supports for Greengrass nucleus lite
3.1.0	Bug fixes and improvements • Adds support for HTTPS network proxy configurations. For more information, see Connect on port 443 or through a network proxy and Enable the core device to trust an HTTPS proxy.
3.0.0	This version of the CloudWatch metrics component expects different configuration parameters than version 2.x. If you use a non-default configuration for version 2.x, and you want to upgrade from v2.x to v3.x, you must update the component's configuration. For more information, see CloudWatch metrics component configuration. New features • Adds support for core devices that run Windows. • Changes the component type from Lambda component to generic component. This component now no longer depends on the legacy subscription router component to create subscriptions. • Adds new `InputTopic` configuration parameter to specify the topic to which the component subscribes to receive messages. • Adds new `OutputTopic` configuration parameter to specify the topic to which the component publishes status responses. • Adds new `PubSubToIoTCore` configuration parameter to specify whether to publish and subscribe to AWS IoT Core MQTT topics. • Adds the new `UseInstaller` configuration parameter that lets you optionally disable the installation script that installs component dependencies. Bug fixes and improvements Adds support for duplicate timestamps in input data.

v2.x

Version	Changes
2.1.8	Version updated for Greengrass nucleus version 2.13.0 release.
2.1.3	Version updated for Greengrass nucleus version 2.11.0 release.
2.1.2	Version updated for Greengrass nucleus version 2.7.0 release.
2.1.1	Version updated for Greengrass nucleus version 2.6.0 release.
2.1.0	New features • Adds support for HTTPS network proxy configurations. For more information, see Connect on port 443 or through a network proxy and Enable the core device to trust an HTTPS proxy.
2.0.8	Bug fixes and improvements • Adds support for duplicate timestamps in input data. • Version updated for Greengrass nucleus version 2.5.0 release.
2.0.7	Version updated for Greengrass nucleus version 2.4.0 release.
2.0.6	Version updated for Greengrass nucleus version 2.3.0 release.
2.0.5	Version updated for Greengrass nucleus version 2.2.0 release.
2.0.4	Version updated for Greengrass nucleus version 2.1.0 release.
2.0.3	Initial version.

See also

- Using Amazon CloudWatch metrics in the *Amazon CloudWatch User Guide*
- PutMetricData in the *Amazon CloudWatch API Reference*

AWS IoT Device Defender

The AWS IoT Device Defender component (`aws.greengrass.DeviceDefender`) notifies administrators about changes in the state of Greengrass core devices. This can help identify unusual behavior that might indicate a compromised device. For more information, see AWS IoT Device Defender in the *AWS IoT Core Developer Guide*.

This component reads system metrics on the core device. Then, it publishes the metrics to AWS IoT Device Defender. For more information about how to read and interpret the metrics that this component reports, see Device metrics document specification in the *AWS IoT Core Developer Guide*.

> **Note**
>
> This component provides similar functionality to the Device Defender connector in AWS IoT Greengrass V1. For more information, see Device Defender connector in the *AWS IoT Greengrass V1 Developer Guide*.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Input data
- Output data
- Local log file
- Licenses
- Changelog

Versions

This component has the following versions:

- 4.0.x
- 3.1.x
- 3.0.x
- 2.0.x

For information about changes in each version of the component, see the changelog.

Type

v4.x - v3.x

This component is a generic component (`aws.greengrass.generic`). The Greengrass nucleus runs the component's lifecycle scripts.

v2.x

This component is a Lambda component (`aws.greengrass.lambda`). The Greengrass nucleus runs this component's Lambda function using the Lambda launcher component.

For more information, see Component types.

Operating system

v4.x - v3.x

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

v2.x

This component can be installed on Linux core devices only.

Requirements

This component has the following requirements:

v4.x - v3.x

- Python version 3.7 installed on the core device and added to the PATH environment variable.
- The following Python libraries, including any dependencies, must be installed and available to the user running the component:
  - AWS IoT Device SDK v2 for Python
  - cbor library. Version 1.0.0 is the latest version that is verified to work with the component.
  - psutil library. Version 5.7.0 is the latest version that is verified to work with the component.

  > **Note**
  >
  > You can set the `UseInstaller` configuration to `true` to install these libraries automatically into the provided Python environment.

- AWS IoT Device Defender configured to use the Detect feature to monitor violations. For more information, see Detect in the *AWS IoT Core Developer Guide*.

v2.x

- Your core device must meet the requirements to run Lambda functions. If you want the core device to run containerized Lambda functions, the device must meet the requirements to do so. For more information, see Lambda function requirements.
- Python version 3.7 installed on the core device and added to the PATH environment variable.
- AWS IoT Device Defender configured to use the Detect feature to monitor violations. For more information, see Detect in the *AWS IoT Core Developer Guide*.
- The psutil library installed on the core device. Version 5.7.0 is the latest version that is verified to work with the component.
- The cbor library installed on the core device. Version 1.0.0 is the latest version that is verified to work with the component.
- To receive output data from this component, you must merge the following configuration update for the legacy subscription router component (`aws.greengrass.LegacySubscriptionRouter`) when you deploy this component. This configuration specifies the topic where this component publishes responses.

Legacy subscription router v2.1.x

```
{
  "subscriptions": {
    "aws-greengrass-device-defender": {
      "id": "aws-greengrass-device-defender",
      "source": "component:aws.greengrass.DeviceDefender",
      "subject": "$aws/things/+/defender/metrics/json",
      "target": "cloud"
    }
  }
}
```

Legacy subscription router v2.0.x

```
{
  "subscriptions": {
    "aws-greengrass-device-defender": {
      "id": "aws-greengrass-device-defender",
      "source": "arn:aws:lambda:region:aws:function:aws-greengrass-device-
defender:version",
      "subject": "$aws/things/+/defender/metrics/json",
      "target": "cloud"
    }
  }
}
```

- Replace *region* with the AWS Region that you use.
- Replace *version* with the version of the Lambda function that this component runs. To find the Lambda function version, you must view the recipe for the version of this component that you want to deploy. Open this component's details page in the AWS IoT Greengrass console, and look for the **Lambda function** key-value pair. This key-value pair contains the name and version of the Lambda function.

> **Important**
>
> You must update the Lambda function version on the legacy subscription router every time you deploy this component. This ensures that you use the correct Lambda function version for the component version that you deploy.

For more information, see Create deployments.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

4.0.0 - 3.0.0

The following table lists the dependencies for versions 4.0.0 to 3.0.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <3.0.0	Soft
Token exchange service	>=0.0.0	Hard

2.0.12 - 2.0.17

The following table lists the dependencies for version 2.0.12 to 2.0.17 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <3.0.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.0.12 - 2.0.16

The following table lists the dependencies for version 2.0.16 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <3.0.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.0.10 - 2.0.11

The following table lists the dependencies for version 2.0.10 and 2.0.11 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.8.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.0.9

The following table lists the dependencies for version 2.0.9 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.7.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft

Dependency	Compatible versions	Dependency type
Token exchange service	^2.0.0	Hard

2.0.8

The following table lists the dependencies for version 2.0.8 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.6.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.0.7

The following table lists the dependencies for version 2.0.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.5.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.0.6

The following table lists the dependencies for version 2.0.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.4.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.0.5

The following table lists the dependencies for version 2.0.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.3.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.0.4

The following table lists the dependencies for version 2.0.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.2.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.0.3

The following table lists the dependencies for version 2.0.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.3 <2.1.0	Hard
Lambda launcher	>=1.0.0	Hard
Lambda runtimes	>=1.0.0	Soft
Token exchange service	>=1.0.0	Hard

For more information about component dependencies, see the component recipe reference.

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

v4.x

`PublishRetryCount`

The amount of times the publish will be retried.

The minimum is 0.

The maximum is 72.

Default: 5

`SampleIntervalSeconds`

(Optional) The amount of time in seconds between each cycle where the component gathers and reports metrics.

The minimum value is 300 seconds (5 minutes).

Default: 300 seconds

`UseInstaller`

(Optional) Boolean value that defines whether to use the installer script to install this component's Python dependencies into the provided Python environment.

Set this value to `true` to automatically install this component's Python dependencies. When set to `false`, you must install the following libraries, including any dependencies, and make them available to the Python environment provided to the user running the component.

- AWS IoT Device SDK v2 for Python
- cbor library. Version 1.0.0 is the latest version that is verified to work with the component.
- psutil library. Version 5.7.0 is the latest version that is verified to work with the component.

Default: `false`

v3.x

`PublishRetryCount`

The amount of times the publish will be retried. This feature is available in version 3.1.1.

The minimum is 0.

The maximum is 72.

Default: 5

`SampleIntervalSeconds`

(Optional) The amount of time in seconds between each cycle where the component gathers and reports metrics.

The minimum value is 300 seconds (5 minutes).

Default: 300 seconds

`UseInstaller`

(Optional) Boolean value that defines whether to use the installer script to install this component's Python dependencies into the provided Python environment.

Set this value to `false` to use a custom script or manually install dependencies. When set to `false`, you must install the following libraries, including any dependencies, and make them available to the Python environment provided to the user running the component.

- AWS IoT Device SDK v2 for Python
- cbor library. Version 1.0.0 is the latest version that is verified to work with the component.
- psutil library. Version 5.7.0 is the latest version that is verified to work with the component.

> **Note**
>
> If you use version 3.0.0 or 3.0.1 of this component on core devices that you configure to use an HTTPS proxy, you must set this value to `false`. The installer script doesn't support operation behind an HTTPS proxy in these versions of this component.

Default: `true`

v2.x

> **Note**
>
> This component's default configuration includes Lambda function parameters. We recommend that you edit only the following parameters to configure this component on your devices.

`lambdaParams`

An object that contains the parameters for this component's Lambda function. This object contains the following information:

`EnvironmentVariables`

An object that contains the Lambda function's parameters. This object contains the following information:

`PROCFS_PATH`

(Optional) The path to the `/proc` folder.

- To run this component in a container, use the default value, `/host-proc`. The component runs in a container by default.
- To run this component in no container mode, specify `/proc` for this parameter.

Default: `/host-proc`. This is the default path where this component mounts the `/proc` folder in the container.

> **Note**
>
> This component has read-only access to this folder.

`SAMPLE_INTERVAL_SECONDS`

(Optional) The amount of time in seconds between each cycle where the component gathers and reports metrics.

The minimum value is 300 seconds (5 minutes).

Default: 300 seconds

`containerMode`

(Optional) The containerization mode for this component. Choose from the following options:

- `GreengrassContainer` – The component runs in an isolated runtime environment inside the AWS IoT Greengrass container.
- `NoContainer` – The component doesn't run in an isolated runtime environment.

  If you specify this option, you must specify `/proc` for the `PROCFS_PATH` environment variable parameter.

Default: `GreengrassContainer`

`containerParams`

(Optional) An object that contains the container parameters for this component. The component uses these parameters if you specify `GreengrassContainer` for `containerMode`.

This object contains the following information:

`memorySize`

(Optional) The amount of memory (in kilobytes) to allocate to the component.

Defaults to 50,000 KB.

`pubsubTopics`

(Optional) An object that contains the topics where the component subscribes to receive messages. You can specify each topic and whether the component subscribes to MQTT topics from AWS IoT Core or local publish/subscribe topics.

This object contains the following information:

`0` – This is an array index as a string.

An object that contains the following information:

`type`

(Optional) The type of publish/subscribe messaging that this component uses to subscribe to messages. Choose from the following options:

- PUB_SUB – Subscribe to local publish/subscribe messages. If you choose this option, the topic can't contain MQTT wildcards. For more information about how to send messages from custom component when you specify this option, see Publish/subscribe local messages.
- `IOT_CORE` – Subscribe to AWS IoT Core MQTT messages. If you choose this option, the topic can contain MQTT wildcards. For more information about how to send messages from custom components when you specify this option, see Publish/subscribe AWS IoT Core MQTT messages.

Default: PUB_SUB

`topic`

(Optional) The topic to which the component subscribes to receive messages. If you specify `IotCore` for `type`, you can use MQTT wildcards (+ and #) in this topic.

Example: Configuration merge update (container mode)

```
{
  "lambdaExecutionParameters": {
    "EnvironmentVariables": {
```

```
      "PROCFS_PATH": "/host_proc"
    }
  },
  "containerMode": "GreengrassContainer"
}
```

Example: Configuration merge update (no container mode)

```
{
  "lambdaExecutionParameters": {
    "EnvironmentVariables": {
      "PROCFS_PATH": "/proc"
    }
  },
  "containerMode": "NoContainer"
}
```

Input data

This component doesn't accept messages as input data.

Output data

This component publishes security metrics to the following reserved topic for AWS IoT Device Defender. This component replaces *coreDeviceName* with the name of the core device when it publishes the metrics.

Topic (AWS IoT Core MQTT): `$aws/things/coreDeviceName/defender/metrics/json`

Example output

```
{
  "header": {
    "report_id": 1529963534,
    "version": "1.0"
  },
  "metrics": {
    "listening_tcp_ports": {
      "ports": [
        {
          "interface": "eth0",
          "port": 24800
```

```
        },
        {
          "interface": "eth0",
          "port": 22
        },
        {
          "interface": "eth0",
          "port": 53
        }
      ],
      "total": 3
    },
    "listening_udp_ports": {
      "ports": [
        {
          "interface": "eth0",
          "port": 5353
        },
        {
          "interface": "eth0",
          "port": 67
        }
      ],
      "total": 2
    },
    "network_stats": {
      "bytes_in": 1157864729406,
      "bytes_out": 1170821865,
      "packets_in": 693092175031,
      "packets_out": 738917180
    },
    "tcp_connections": {
      "established_connections":{
        "connections": [
          {
            "local_interface": "eth0",
            "local_port": 80,
            "remote_addr": "192.168.0.1:8000"
          },
          {
            "local_interface": "eth0",
            "local_port": 80,
            "remote_addr": "192.168.0.1:8000"
          }
```

```
        ],
        "total": 2
      }
    }
  }
}
```

For more information about the metrics that this component reports, see Device metrics document specification in the *AWS IoT Core Developer Guide*.

Local log file

This component uses the following log file.

Linux

```
/greengrass/v2/logs/aws.greengrass.DeviceDefender.log
```

Windows

```
C:\greengrass\v2\logs\aws.greengrass.DeviceDefender.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/aws.greengrass.DeviceDefender.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\aws.greengrass.DeviceDefender.log -Tail 10 -
  Wait
  ```

Licenses

This component is released under the Greengrass Core Software License Agreement.

Changelog

The following table describes the changes in each version of the component.

v4.x - v3.x

Version	Changes
4.0.0	New features • The `UseInstaller` configuration now defaults to `false`.
3.1.1	Bug fixes and improvements • Adds retries for client connection when the connection fails to recover after a network outage. • Adds a configurable retry for publishing metrics.
3.1.0	Bug fixes and improvements • Adds support for HTTPS network proxy configurations. For more information, see Connect on port 443 or through a network proxy and Enable the core device to trust an HTTPS proxy.
3.0.1	Fixes an issue with how the component calculates delta values for metrics.
3.0.0	**Warning** This version is no longer available. The improvements in this version are available in later versions of this component. Initial version.

v2.x

Version	Changes
2.0.17	Version updated for Greengrass nucleus version 2.14.0 release.
2.0.16	Version updated for Greengrass nucleus version 2.13.0 release.
2.0.11	Version updated for Greengrass nucleus version 2.11.0 release.
2.0.10	Version updated for Greengrass nucleus version 2.7.0 release.
2.0.9	Version updated for Greengrass nucleus version 2.6.0 release.
2.0.8	Version updated for Greengrass nucleus version 2.5.0 release.
2.0.7	Version updated for Greengrass nucleus version 2.4.0 release.
2.0.6	Version updated for Greengrass nucleus version 2.3.0 release.
2.0.5	Version updated for Greengrass nucleus version 2.2.0 release.
2.0.4	Version updated for Greengrass nucleus version 2.1.0 release.
2.0.3	Initial version.

Disk spooler

The disk spooler component (`aws.greengrass.DiskSpooler`) offers a persistent storage option for messages spooled from Greengrass core devices to AWS IoT Core. This component will store these outbound messages on disk.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies

- Usage
- Local log file
- Changelog

Versions

This component has the following versions:

- 1.0.x

Type

This component is a plugin component (`aws.greengrass.plugin`). The Greengrass nucleus runs this component in the same Java Virtual Machine (JVM) as the nucleus. The nucleus restarts when you change this component's version on the core device.

This component uses the same log file as the Greengrass nucleus. For more information, see Monitor AWS IoT Greengrass logs.

For more information, see Component types.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

Requirements

This component has the following requirements:

- `storageType` should be set to `Disk` to use this component. You can set this in the Greengrass nucleus configuration.
- `maxSizeInBytes` must not be configured to be greater than the available space on the device. You can set this in the Greengrass nucleus configuration.
- The disk spooler component is supported to run in a VPC.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

1.0.7

The following table lists the dependencies for version 1.0.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.11.0 <2.17.0	Hard

1.0.

The following table lists the dependencies for version 1.0.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.11.0 <2.16.0	Hard

1.0.5

The following table lists the dependencies for version 1.0.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.11.0 <2.15.0	Hard

1.0.4

The following table lists the dependencies for version 1.0.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.11.0 <2.14.0	Hard

1.0.1 – 1.0.3

The following table lists the dependencies for versions 1.0.1 to 1.0.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.11.0 <2.13.0	Hard

1.0.0

The following table lists the dependencies for version 1.0.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.11.0 <2.12.0	Hard

For more information about component dependencies, see the component recipe reference.

Usage

To use the disk spooler component, `aws.greengrass.DiskSpooler` must be deployed.

To configure and use this component, you must set the `pluginName` to `aws.greengrass.DiskSpooler`.

Local log file

This component uses the same log file as the Greengrass nucleus component.

Linux

```
/greengrass/v2/logs/greengrass.log
```

Windows

```
C:\greengrass\v2\logs\greengrass.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
  ```

Changelog

The following table describes the changes in each version of the component.

Version	Changes
1.0.9	Updates the component version for the Greengrass nucleus version 2.18.0 release.
1.0.8	Updates the component version for the Greengrass nucleus version 2.17.0 release.
1.0.7	Version updated for Greengrass nucleus version 2.16.0 release.
1.0.6	Version updated for Greengrass nucleus version 2.15.0 release.
1.0.5	Version updated for Greengrass nucleus version 2.14.0 release.

Version	Changes
1.0.4	Bug fixes and improvements General bug fixes.
1.0.3	Bug fixes and improvements Improves performance by reusing database connections.
1.0.2	Bug fixes and improvements Fixes an issue where the MQTT message format field isn't persisted in certain cases.
1.0.1	Version updated for Greengrass nucleus version 2.12.0 release.
1.0.0	Initial version.

Docker application manager

The Docker application manager component (`aws.greengrass.DockerApplicationManager`) enables AWS IoT Greengrass to download Docker images from public image registries and private registries hosted on Amazon Elastic Container Registry (Amazon ECR). It also enables AWS IoT Greengrass to manage credentials automatically to securely download images from private repositories in Amazon ECR.

When you develop a custom component that runs a Docker container, include the Docker application manager as a dependency to download the Docker images that are specified as artifacts in your component. For more information, see Run a Docker container.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration

- Local log file
- Changelog
- See also

Versions

This component has the following versions:

- 2.0.x

Type

This component is a generic component (`aws.greengrass.generic`). The Greengrass nucleus runs the component's lifecycle scripts.

For more information, see Component types.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

Requirements

This component has the following requirements:

- Docker Engine 1.9.1 or later installed on the Greengrass core device. Versions up to and including 29.6.0, such as 20.10, 24.0, 25.0, and 27.5, work with the AWS IoT Greengrass Core software. You must install Docker directly on the core device before you deploy components that run Docker containers.
- The Docker daemon started and running on the core device before you deploy this component.
- Docker images stored in one of the following supported image sources:
  - Public and private image repositories in Amazon Elastic Container Registry (Amazon ECR)
  - Public Docker Hub repository
  - Public Docker Trusted Registry

- Docker images included as artifacts in your custom Docker container components. Use the following URI formats to specify your Docker images:
  - Private Amazon ECR image: `docker:`*`account-id`*`.dkr.ecr.`*`region`*`.amazonaws.com/`*`repository`*`/`*`image`*`[`*`:tag|@digest`*`]`
  - Public Amazon ECR image: `docker:public.ecr.aws/`*`repository`*`/`*`image`*`[`*`:tag|@digest`*`]`
  - Public Docker Hub image: `docker:`*`name`*`[`*`:tag|@digest`*`]`

  For more information, see Run a Docker container.

  > **Note**
  >
  > If you don't specify the image tag or image digest in the artifact URI for an image, then the Docker application manager pulls the latest available version of that image when you deploy your custom Docker container component. To ensure that all of your core devices run the same version of an image, we recommend that you include the image tag or image digest in the artifact URI.

- The system user that runs a Docker container component must have root or administrator permissions, or you must configure Docker to run it as a non-root or non-admistrator user.
  - On Linux devices, you can add a user to the `docker` group to call `docker` commands without `sudo`.
  - On Windows devices, you can add a user to the `docker-users` group to call `docker` commands without adminstrator privileges.

  Linux or Unix

  To add `ggc_user`, or the non-root user that you use to run Docker container components, to the `docker` group, run the following command.

  ```
  sudo usermod -aG docker ggc_user
  ```

  For more information, see Manage Docker as a non-root user.

  Windows Command Prompt (CMD)

  To add `ggc_user`, or the user that you use to run Docker container components, to the `docker-users` group, run the following command as an administrator.

```
net localgroup docker-users ggc_user /add
```

Windows PowerShell

To add `ggc_user`, or the user that you use to run Docker container components, to the `docker-users` group, run the following command as an administrator.

```
Add-LocalGroupMember -Group docker-users -Member ggc_user
```

- If you configure the AWS IoT Greengrass Core software to use a network proxy, you must configure Docker to use the same proxy server.
- If your Docker images are stored in an Amazon ECR private registry, then you must include the token exchange service component as a dependency in the Docker container component. Also, the Greengrass device role must allow the `ecr:GetAuthorizationToken`, `ecr:BatchGetImage`, and `ecr:GetDownloadUrlForLayer` actions, as shown in the following example IAM policy.

  JSON

  ```
  {
    "Version":"2012-10-17",
    "Statement": [
      {
        "Action": [
          "ecr:GetAuthorizationToken",
          "ecr:BatchGetImage",
          "ecr:GetDownloadUrlForLayer"
        ],
        "Resource": [
          "*"
        ],
        "Effect": "Allow"
      }
    ]
  }
  ```

- The docker application manager component is supported to run in a VPC. To deploy this component in a VPC, the following is required.

- The docker application manager component must have connectivity to download images. For example, if you use ECR, you must have connectivity to the following endpoints.
  - `*.dkr.ecr.region.amazonaws.com` (VPC endpoint `com.amazonaws.region.ecr.dkr`)
  - `api.ecr.region.amazonaws.com` (VPC endpoint `com.amazonaws.region.ecr.api`)

Endpoints and ports

This component must be able to perform outbound requests to the following endpoints and ports, in addition to endpoints and ports required for basic operation. For more information, see Allow device traffic through a proxy or firewall.

Endpoint	Port	Required	Description
`ecr.region.amazonaws.com`	443	No	Required if you download Docker images from Amazon ECR.

`hub.docker.com`
 	
`registry.hub.docker.com/v1`
	443	No	Required if you download Docker images from Docker Hub.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

2.0.15

The following table lists the dependencies for version 2.0.15 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.17.0	Soft

2.0.14

The following table lists the dependencies for version 2.0.14 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.16.0	Soft

2.0.13

The following table lists the dependencies for version 2.0.13 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.15.0	Soft

2.0.12

The following table lists the dependencies for version 2.0.12 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.14.0	Soft

2.0.11

The following table lists the dependencies for version 2.0.11 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.13.0	Soft

2.0.10

The following table lists the dependencies for version 2.0.10 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.12.0	Soft

2.0.9

The following table lists the dependencies for version 2.0.9 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.11.0	Soft

2.0.8

The following table lists the dependencies for version 2.0.8 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.10.0	Soft

2.0.7

The following table lists the dependencies for version 2.0.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.9.0	Soft

2.0.6

The following table lists the dependencies for version 2.0.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.8.0	Soft

2.0.5

The following table lists the dependencies for version 2.0.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.7.0	Soft

2.0.4

The following table lists the dependencies for version 2.0.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.6.0	Soft

2.0.3

The following table lists the dependencies for version 2.0.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.5.0	Soft

2.0.2

The following table lists the dependencies for version 2.0.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.4.0	Soft

2.0.1

The following table lists the dependencies for version 2.0.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.3.0	Soft

2.0.0

The following table lists the dependencies for version 2.0.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.2.0	Soft

For more information about component dependencies, see the component recipe reference.

Configuration

This component doesn't have any configuration parameters.

Local log file

This component uses the same log file as the Greengrass nucleus component.

Linux

```
/greengrass/v2/logs/greengrass.log
```

Windows

```
C:\greengrass\v2\logs\greengrass.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
  ```

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.0.17	Updates the component version for the Greengrass nucleus version 2.18.0 release.
2.0.16	Updates the component version for the Greengrass nucleus version 2.17.0 release.
2.0.15	Version updated for Greengrass nucleus version 2.16.0 release.
2.0.14	Version updated for Greengrass nucleus version 2.15.0 release.

Version	Changes
2.0.13	Version updated for Greengrass nucleus version 2.14.0 release.
2.0.12	Version updated for Greengrass nucleus version 2.13.0 release.
2.0.11	Version updated for Greengrass nucleus version 2.12.0 release.
2.0.10	Version updated for Greengrass nucleus version 2.11.0 release.
2.0.9	Version updated for Greengrass nucleus version 2.10.0 release.
2.0.8	Version updated for Greengrass nucleus version 2.9.0 release.
2.0.7	Version updated for Greengrass nucleus version 2.8.0 release.
2.0.6	Version updated for Greengrass nucleus version 2.7.0 release.
2.0.5	Version updated for Greengrass nucleus version 2.6.0 release.
2.0.4	Version updated for Greengrass nucleus version 2.5.0 release.
2.0.3	Version updated for Greengrass nucleus version 2.4.0 release.
2.0.2	Version updated for Greengrass nucleus version 2.3.0 release.
2.0.1	Version updated for Greengrass nucleus version 2.2.0 release.
2.0.0	Initial version.

See also

- Run a Docker container

Edge connector for Kinesis Video Streams

The edge connector for Kinesis Video Streams component (`aws.iot.EdgeConnectorForKVS`) reads video feeds from local cameras and publishes the streams to Kinesis Video Streams. You can configure this component to read video feeds from Internet Protocol (IP) cameras using Real Time

Streaming Protocol (RTSP). Then, you can set up dashboards in Amazon Managed Grafana or local Grafana servers to monitor and interact with the video streams.

You can integrate this component with AWS IoT TwinMaker to display and control video streams in Grafana dashboards. AWS IoT TwinMaker is an AWS service that enables you to build operational digital twins of physical systems. You can use AWS IoT TwinMaker to visualize data from sensors, cameras, and enterprise applications for you to track your physical factories, buildings, or industrial plants. You can also use this data to monitor operations, diagnose errors, and repair errors. For more information, see What is AWS IoT TwinMaker? in the *AWS IoT TwinMaker User Guide*.

This component stores its configuration in AWS IoT SiteWise, which is an AWS service that models and stores industrial data. In AWS IoT SiteWise, *assets* represent objects such as devices, equipment, or groups of other objects. To configure and use this component, you create an AWS IoT SiteWise asset for each Greengrass core device and for each IP camera connected to each core device. Each asset has properties that you configure to control features, such as live streaming, on-demand upload, and local caching. To specify the URL for each camera, you create a secret in AWS Secrets Manager that contains the URL of the camera. If the camera requires authentication, you also specify a user name and password in the URL. Then, you specify that secret in an asset property for the IP camera.

This component uploads each camera's video stream to a Kinesis video stream. You specify the name of the destination Kinesis video stream in the AWS IoT SiteWise asset configuration for each camera. If the Kinesis video stream doesn't exist, this component creates it for you.

AWS IoT TwinMaker provides a script that you can run to create these AWS IoT SiteWise assets and Secrets Manager secrets. For more information about how to create these resources, and how to install, configure, and use this component, see AWS IoT TwinMaker video integration in the *AWS IoT TwinMaker User Guide*.

Note

The edge connector for Kinesis Video Streams component is available only in the following AWS Regions:

- US East (N. Virginia)
- US West (Oregon)
- Europe (Frankfurt)
- Europe (Ireland)

- Asia Pacific (Singapore)
- Asia Pacific (Tokyo)
- Asia Pacific (Seoul)
- Asia Pacific (Sydney)
- Asia Pacific (Mumbai)
- China (Beijing)

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Licenses
- Usage
- Local log file
- Changelog
- See also

Versions

This component has the following versions:

- 1.0.x

Type

This component is a generic component (`aws.greengrass.generic`). The Greengrass nucleus runs the component's lifecycle scripts.

For more information, see Component types.

Operating system

This component can be installed on Linux core devices only.

Requirements

This component has the following requirements:

- You can deploy this component to only single core devices, because the component configuration must be unique for each core device. You can't deploy this component to groups of core devices.
- GStreamer 1.18.4 or later installed on the core device. For more information, see Installing GStreamer.

  On a device with `apt`, you can run the following commands to install GStreamer.

  ```
  sudo apt install -y libgstreamer1.0-dev libgstreamer-plugins-base1.0-dev
   gstreamer1.0-plugins-base-apps
  sudo apt install -y gstreamer1.0-libav
  sudo apt install -y gstreamer1.0-plugins-bad gstreamer1.0-plugins-good gstreamer1.0-
  plugins-ugly gstreamer1.0-tools
  ```

- An AWS IoT SiteWise asset for each core device. This AWS IoT SiteWise asset represents the core device. For more information about how to create this asset, see AWS IoT TwinMaker video integration in the *AWS IoT TwinMaker User Guide*.
- An AWS IoT SiteWise asset for each IP camera that you connect to each core device. These AWS IoT SiteWise assets represent the cameras that stream video to each core device. Each camera's asset must be associated to the asset for the core device that connects to the camera. Camera assets have properties that you can configure to specify a Kinesis video stream, an authentication secret, and video streaming parameters. For more information about how to create and configure camera assets, see AWS IoT TwinMaker video integration in the *AWS IoT TwinMaker User Guide*.
- An AWS Secrets Manager secret for each IP camera. This secret must define a key-value pair, where the key is `RTSPStreamUrl`, and the value is the URL for the camera. If the camera requires authentication, include the user name and password in this URL. You can use a script to create a secret when you create the resources that this component requires. For more information, see AWS IoT TwinMaker video integration in the *AWS IoT TwinMaker User Guide*.

You can also use the Secrets Manager console and API to create additional secrets. For more information, see Create a secret in the *AWS Secrets Manager User Guide*.

- The Greengrass token exchange role must allow the following AWS Secrets Manager, AWS IoT SiteWise, and Kinesis Video Streams actions, as shown in the following example IAM policy.

> **Note**
>
> This example policy allows the device to get the value of secrets named **`IPCamera1Url`** and **`IPCamera2Url`**. When you configure each IP camera, you specify a secret that contains the URL for that camera. If the camera requires authentication, you also specify a user name and password in the URL. The core device's token exchange role must allow access to the secret for each IP camera to connect.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Action": [
        "secretsmanager:GetSecretValue"
      ],
      "Effect": "Allow",
      "Resource": [
      "arn:aws:secretsmanager:us-east-1:123456789012:secret:IPCamera1Url",
  "arn:aws:secretsmanager:us-east-1:123456789012:secret:IPCamera2Url"
      ]
    },
    {
      "Action": [
        "iotsitewise:BatchPutAssetPropertyValue",
        "iotsitewise:DescribeAsset",
        "iotsitewise:DescribeAssetModel",
        "iotsitewise:DescribeAssetProperty",
        "iotsitewise:GetAssetPropertyValue",
        "iotsitewise:ListAssetRelationships",
        "iotsitewise:ListAssets",
        "iotsitewise:ListAssociatedAssets",
        "kinesisvideo:CreateStream",
```

```
        "kinesisvideo:DescribeStream",
        "kinesisvideo:GetDataEndpoint",
        "kinesisvideo:PutMedia",
        "kinesisvideo:TagStream"
      ],
      "Effect": "Allow",
      "Resource": [
        "*"
      ]
    }
  ]
}
```

> **Note**
>
> If you use a customer managed AWS Key Management Service key to encrypt secrets, the device role must also allow the `kms:Decrypt` action.

Endpoints and ports

This component must be able to perform outbound requests to the following endpoints and ports, in addition to endpoints and ports required for basic operation. For more information, see Allow device traffic through a proxy or firewall.

Endpoint	Port	Required	Description
`kinesisvideo.` *`region`*`.amazonaws.com`	443	Yes	Upload data to Kinesis Video Streams.
`data.iotsitewise.` *`region`*`.amazonaws.com`	443	Yes	Publish video stream metadata

Endpoint	Port	Required	Description
			to AWS IoT SiteWise.
`secretsmanager.` *region* `.amazonaws.com`	443	Yes	Download camera URL secrets to the core device.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

The following table lists the dependencies for versions 1.0.0 to 1.0.5 of this component.

Dependency	Compatible versions	Dependency type
Token exchange service	>=2.0.3	Hard
Stream manager	>=2.0.9	Hard

For more information about component dependencies, see the component recipe reference.

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

`SiteWiseAssetIdForHub`

The ID of the AWS IoT SiteWise asset that represents this core device. For more information about how to create this asset and use it to interact with this component, see AWS IoT TwinMaker video integration in the *AWS IoT TwinMaker User Guide*.

Example: Configuration merge update

```
{
  "SiteWiseAssetIdForHub": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
}
```

Licenses

This component includes the following third-party software/licensing:

- Quartz Job Scheduler / Apache License 2.0
- Java bindings for GStreamer 1.x / GNU Lesser General Public License v3.0

Usage

To configure and interact with this component, you can set properties on the AWS IoT SiteWise assets that represent the core device and the IP cameras where it connects. You can also visualize and interact with video streams in Grafana dashboards through AWS IoT TwinMaker. For more information, see AWS IoT TwinMaker video integration in the *AWS IoT TwinMaker User Guide*.

Local log file

This component uses the following log file.

```
/greengrass/v2/logs/aws.iot.EdgeConnectorForKVS.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* with the path to the AWS IoT Greengrass root folder.

  ```
  sudo tail -f /greengrass/v2/logs/aws.iot.EdgeConnectorForKVS.log
  ```

Changelog

The following table describes the changes in each version of the component.

Version	Changes
1.0.5	General bug fixes and improvements.
1.0.4	Bug fixes and improvements • Fixes an issue that caused live uploading to stop.
1.0.3	General bug fixes and improvements.
1.0.1	General bug fixes and improvements.
1.0.0	Initial version.

See also

- What is AWS IoT TwinMaker? in the *AWS IoT TwinMaker User Guide*
- AWS IoT TwinMaker video integration in the *AWS IoT TwinMaker User Guide*
- What is AWS IoT SiteWise? in the *AWS IoT SiteWise User Guide*
- Updating attribute values in the *AWS IoT SiteWise User Guide*
- What is AWS Secrets Manager? in the *AWS Secrets Manager User Guide*
- Create and manage secrets in the *AWS Secrets Manager User Guide*

Greengrass CLI

The Greengrass CLI component (`aws.greengrass.Cli`) provides a local command-line interface that you can use on core devices to develop and debug components locally. The Greengrass CLI lets you create local deployments and restart components on the core device, for example.

You can install this component when you install the AWS IoT Greengrass Core software. For more information, see Tutorial: Getting started with AWS IoT Greengrass V2.

Important

We recommend that you use this component in only development environments, not production environments. This component provides access to information and operations that you typically won't need in a production environment. Follow the principle of least privilege by deploying this component to only core devices where you need it.

After you install this component, run the following command to view its help documentation. When this component installs, it adds a symbolic link to `greengrass-cli` in the */greengrass/v2*`/bin` folder. You can run the Greengrass CLI from this path or add it to your `PATH` environment variable to run `greengrass-cli` without its absolute path.

Linux or Unix

```
/greengrass/v2/bin/greengrass-cli help
```

Windows

```
C:\greengrass\v2\bin\greengrass-cli help
```

The following command restarts a component named `com.example.HelloWorld`, for example.

Linux or Unix

```
sudo /greengrass/v2/bin/greengrass-cli component restart --names
 "com.example.HelloWorld"
```

Windows

```
C:\greengrass\v2\bin\greengrass-cli component restart --names
 "com.example.HelloWorld"
```

For more information, see Greengrass Command Line Interface.

Topics

- Versions

- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Local log file
- Changelog

Versions

This component has the following versions:

- 2.18.x
- 2.17.x
- 2.16.x
- 2.15.x
- 2.14.x
- 2.13.x
- 2.12.x
- 2.11.x
- 2.10.x
- 2.9.x
- 2.8.x
- 2.7.x
- 2.6.x
- 2.5.x
- 2.4.x
- 2.3.x
- 2.2.x
- 2.1.x
- 2.0.x

Type

This component is a plugin component (`aws.greengrass.plugin`). The Greengrass nucleus runs this component in the same Java Virtual Machine (JVM) as the nucleus. The nucleus restarts when you change this component's version on the core device.

This component uses the same log file as the Greengrass nucleus. For more information, see Monitor AWS IoT Greengrass logs.

For more information, see Component types.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

Requirements

This component has the following requirements:

- You must be authorized to use the Greengrass CLI to interact with the AWS IoT Greengrass Core software. Do one of the following to use the Greengrass CLI:
  - Use the system user that runs the AWS IoT Greengrass Core software.
  - Use a user with root or adminstrative permissions. On Linux core devices, you can use `sudo` to gain root permissions.
  - Use a system user that's in a group that you specify in the `AuthorizedPosixGroups` or `AuthorizedWindowsGroups` configuration parameters when you deploy the component. For more information, see Greengrass CLI component configuration.
- The Greengrass CLI component is supported to run in a VPC.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the

component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

2.18.0 - 2.18.4

The following table lists the dependencies for versions 2.18.0 through 2.18.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.12.0 <2.19.0	Soft

2.17.0

The following table lists the dependencies for version 2.17.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.12.0 <2.18.0	Soft

2.16.0

The following table lists the dependencies for version 2.16.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.12.0 <2.17.0	Soft

2.15.1

The following table lists the dependencies for version 2.15.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.12.0 <2.16.0	Soft

2.15.0

The following table lists the dependencies for version 2.15.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.12.0 <2.16.0	Soft

2.14.0 – 2.14.3

The following table lists the dependencies for versions 2.14.0 and 2.14.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.12.0 <2.15.0	Soft

2.13.0

The following table lists the dependencies for version 2.13.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.12.0 <2.14.0	Soft

2.12.0 – 2.12.6

The following table lists the dependencies for version 2.12.0 through 2.12.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.12.0 <2.13.0	Soft

2.11.0 – 2.11.3

The following table lists the dependencies for versions 2.11.0 through 2.11.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.11.0 <2.12.0	Soft

2.10.0 – 2.10.3

The following table lists the dependencies for versions 2.10.0 through 2.10.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.11.0	Soft

2.9.0 – 2.9.6

The following table lists the dependencies for versions 2.9.0 through 2.9.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.10.0	Soft

2.8.0 – 2.8.1

The following table lists the dependencies for version 2.8.0 and 2.8.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.9.0	Soft

2.7.0

The following table lists the dependencies for version 2.7.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.8.0	Soft

2.6.0

The following table lists the dependencies for version 2.6.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.7.0	Soft

2.5.0 – 2.5.6

The following table lists the dependencies for versions 2.5.0 through 2.5.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.6.0	Soft

2.4.0

The following table lists the dependencies for version 2.4.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.5.0	Soft

2.3.0

The following table lists the dependencies for version 2.3.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.4.0	Soft

2.2.0

The following table lists the dependencies for version 2.2.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.3.0	Soft

2.1.0

The following table lists the dependencies for version 2.1.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.2.0	Soft

2.0.x

The following table lists the dependencies for version 2.0.x of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.1.0	Soft

Note

The minimum compatible version of the Greengrass nucleus corresponds to the patch version of the Greengrass CLI component.

For more information about component dependencies, see the component recipe reference.

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

2.5.x - 2.14.x

`AuthorizedPosixGroups`

(Optional) A string that contains a comma-separated list of system groups. You authorize these system groups to use the Greengrass CLI to interact with the AWS IoT Greengrass Core software. You can specify group names or group IDs. For example, `group1,1002,group3` authorizes three system groups (`group1`, `1002`, and `group3`) to use the Greengrass CLI.

If you don't specify any groups to authorize, you can use the Greengrass CLI as the root user (`sudo`) or as the system user that runs the AWS IoT Greengrass Core software.

`AuthorizedWindowsGroups`

(Optional) A string that contains a comma-separated list of system groups. You authorize these system groups to use the Greengrass CLI to interact with the AWS IoT Greengrass Core software. You can specify group names or group IDs. For example, `group1,1002,group3` authorizes three system groups (`group1`, `1002`, and `group3`) to use the Greengrass CLI.

If you don't specify any groups to authorize, you can use the Greengrass CLI as an administrator or as the system user that runs the AWS IoT Greengrass Core software.

Example: Configuration merge update

The following example configuration specifies to authorize three POSIX system groups (`group1`, `1002`, and `group3`) and two Windows user groups (`Device Operators` and `QA Engineers`) to use the Greengrass CLI.

```
{
  "AuthorizedPosixGroups": "group1,1002,group3",
  "AuthorizedWindowsGroups": "Device Operators,QA Engineers"
}
```

2.4.x - 2.0.x

`AuthorizedPosixGroups`

(Optional) A string that contains a comma-separated list of system groups. You authorize these system groups to use the Greengrass CLI to interact with the AWS IoT Greengrass Core software. You can specify group names or group IDs. For example, `group1,1002,group3` authorizes three system groups (`group1`, `1002`, and `group3`) to use the Greengrass CLI.

If you don't specify any groups to authorize, you can use the Greengrass CLI as the root user (`sudo`) or as the system user that runs the AWS IoT Greengrass Core software.

Example: Configuration merge update

The following example configuration specifies to authorize three system groups (`group1`, `1002`, and `group3`) to use the Greengrass CLI.

```
{
  "AuthorizedPosixGroups": "group1,1002,group3"
}
```

Local log file

This component uses the same log file as the Greengrass nucleus component.

Linux

```
/greengrass/v2/logs/greengrass.log
```

Windows

```
C:\greengrass\v2\logs\greengrass.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
  ```

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.18.4	Updates the component version for the Greengrass nucleus version 2.18.4 release.
2.18.3	Updates the component version for the Greengrass nucleus version 2.18.3 release.
2.18.2	Updates the component version for the Greengrass nucleus version 2.18.2 release.
2.18.1	Updates the component version for the Greengrass nucleus version 2.18.1 release.
2.18.0	Updates the component version for the Greengrass nucleus version 2.18.0 release.
2.17.0	Updates the component version for the Greengrass nucleus version 2.17.0 release.
2.16.1	Version updated for Greengrass nucleus version 2.16.1 release.
2.16.0	Version updated for Greengrass nucleus version 2.16.0 release.
2.15.1	Version updated for Greengrass nucleus version 2.15.1 release.
2.15.0	Bug fixes and improvements • Adds a fix so that when the auth configuration changes, the component won't reinstall but will still restart. • General bug fixes and improvements.
2.14.3	Version updated for Greengrass nucleus version 2.14.3 release.
2.14.2	Version updated for Greengrass nucleus version 2.14.2 release.
2.14.1	Version updated for Greengrass nucleus version 2.14.1 release.

Version	Changes
2.14.0	**Warning** This version is no longer available. The improvements in this version are available in later versions of this component. Bug fixes and improvements • Validate deployment target parameter in the cli command.
2.13.0	Version updated for Greengrass nucleus version 2.13.0 release.
2.12.6	Version updated for Greengrass nucleus version 2.12.6 release.
2.12.5	Version updated for Greengrass nucleus version 2.12.5 release.
2.12.4	Version updated for Greengrass nucleus version 2.12.4 release.
2.12.3	**Warning** This version is no longer available. The improvements in this version are available in later versions of this component. Version updated for Greengrass nucleus version 2.12.3 release.
2.12.2	Version updated for Greengrass nucleus version 2.12.2 release.
2.12.1	Version updated for Greengrass nucleus version 2.12.1 release.
2.12.0	Version updated for Greengrass nucleus version 2.12.0 release.
2.11.3	Version updated for Greengrass nucleus version 2.11.3 release.
2.11.2	Version updated for Greengrass nucleus version 2.11.2 release.
2.11.1	Version updated for Greengrass nucleus version 2.11.1 release.

Version	Changes
2.11.0	New features • Enables you to cancel a local deployment. • Enables you to configure a failure handling policy for a local deployment. • Improves detailed deployment status reporting.
2.10.3	Version updated for Greengrass nucleus version 2.10.3 release.
2.10.2	Version updated for Greengrass nucleus version 2.10.2 release.
2.10.1	Version updated for Greengrass nucleus version 2.10.1 release.
2.10.0	Version updated for Greengrass nucleus version 2.10.0 release.
2.9.6	Version updated for Greengrass nucleus version 2.9.6 release.
2.9.5	Version updated for Greengrass nucleus version 2.9.5 release.
2.9.4	Version updated for Greengrass nucleus version 2.9.4 release.
2.9.3	Version updated for Greengrass nucleus version 2.9.3 release.
2.9.2	Version updated for Greengrass nucleus version 2.9.2 release.
2.9.1	Version updated for Greengrass nucleus version 2.9.1 release.
2.9.0	Version updated for Greengrass nucleus version 2.9.0 release.
2.8.1	Version updated for Greengrass nucleus version 2.8.1 release.
2.8.0	Version updated for Greengrass nucleus version 2.8.0 release.
2.7.0	Version updated for Greengrass nucleus version 2.7.0 release.

Version	Changes
2.6.0	New features • Adds support for custom components to call interprocess communication (IPC) operations that the Greengrass CLI uses. You can use these IPC operations to manage local deployments, view component details, and generate a password that you can use to sign in to the local debug console. For more information, see IPC: Manage local deployments and components. Bug fixes and improvements • Additional minor fixes and improvements.
2.5.6	Version updated for Greengrass nucleus version 2.5.6 release.
2.5.5	Version updated for Greengrass nucleus version 2.5.5 release.
2.5.4	Version updated for Greengrass nucleus version 2.5.4 release.
2.5.3	Version updated for Greengrass nucleus version 2.5.3 release.
2.5.2	Version updated for Greengrass nucleus version 2.5.2 release.
2.5.1	Version updated for Greengrass nucleus version 2.5.1 release.
2.5.0	New features • Adds support for core devices that run Windows. • Adds the new `AuthorizedWindowsGroups` configuration parameter that you can specify to authorize system groups to use the Greengrass CLI on Windows devices. • Adds the `windowsUser` parameter for local deployments. You can use this parameter specify the user to use to run components on a Windows core device.

Version	Changes
2.4.0	New features • Adds support for system resource limits. When you create a local deployment, you can configure the maximum amount of CPU and RAM usage that each component's processes can use on the core device. For more information, see Configure system resource limits for components and the deployment create command.
2.3.0	Version updated for Greengrass nucleus version 2.3.0 release.
2.2.0	Version updated for Greengrass nucleus version 2.2.0 release.
2.1.0	Version updated for Greengrass nucleus version 2.1.0 release.
2.0.5	Version updated for Greengrass nucleus version 2.0.5 release.
2.0.4	Version updated for Greengrass nucleus version 2.0.4 release.
2.0.3	Initial version.

IP detector

The IP detector component (`aws.greengrass.clientdevices.IPDetector`) does the following:

- Monitors the Greengrass core device's network connectivity information. This information includes the core device's network endpoints and the port where an MQTT broker operates.
- Updates the core device's connectivity information in the AWS IoT Greengrass cloud service.

Client devices can use Greengrass cloud discovery to retrieve associated core devices' connectivity information. Then, client devices can try to connect to each core device until they successfully connect.

> **Note**
>
> Client devices are local IoT devices that connect to a Greengrass core device to send MQTT messages and data to process. For more information, see Interact with local IoT devices.

The IP detector component replaces a core device's existing connectivity information with the information it detects. Because this component removes existing information, you can either use the IP detector component, or manually manage connectivity information.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Local log file
- Changelog

Versions

This component has the following versions:

- 2.2.x
- 2.1.x
- 2.0.x

Type

This component is a plugin component (`aws.greengrass.plugin`). The Greengrass nucleus runs this component in the same Java Virtual Machine (JVM) as the nucleus. The nucleus restarts when you change this component's version on the core device.

This component uses the same log file as the Greengrass nucleus. For more information, see Monitor AWS IoT Greengrass logs.

For more information, see Component types.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

Requirements

This component has the following requirements:

- The Greengrass service role must be associated to your AWS account and allow the `iot:GetThingShadow` and `iot:UpdateThingShadow` permissions.
- The core device's AWS IoT policy must allow the `greengrass:UpdateConnectivityInfo` permission. For more information, see AWS IoT policies for data plane operations and Minimal AWS IoT policy to support client devices.
- If you configure the core device's MQTT broker component to use a port other than the default port 8883, you must use IP detector v2.1.0 or later. Configure it to report the port where the broker operates.
- If you have a complex network setup, the IP detector component might not be able to identify the endpoints where client devices can connect to the core device. If the IP detector component can't manage the endpoints, you must manually manage the core device endpoints instead. For example, if the core device is behind a router that forwards the MQTT broker port to it, you must specify the router's IP address as an endpoint for the core device. For more information, see Manage core device endpoints.
- The IP detector component is supported to run in a VPC.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of

the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

2.2.3

The following table lists the dependencies for version 2.2.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.2.0 <2.17.0	Soft

2.2.2

The following table lists the dependencies for version 2.2.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.2.0 <2.16.0	Soft

2.2.1

The following table lists the dependencies for version 2.2.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.2.0 <2.15.0	Soft

2.2.0

The following table lists the dependencies for version 2.2.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.2.0 <2.14.0	Soft

2.1.8 – 2.1.9

The following table lists the dependencies for versions 2.1.8 and 2.1.9 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.2.0 <2.13.0	Soft

2.1.7

The following table lists the dependencies for version 2.1.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.2.0 <2.12.0	Soft

2.1.6

The following table lists the dependencies for version 2.1.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.2.0 <2.11.0	Soft

2.1.5

The following table lists the dependencies for version 2.1.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.2.0 <2.10.0	Soft

2.1.4

The following table lists the dependencies for version 2.1.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.2.0 <2.9.0	Soft

2.1.3

The following table lists the dependencies for version 2.1.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.2.0 <2.8.0	Soft

2.1.2

The following table lists the dependencies for version 2.1.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.2.0 <2.7.0	Soft

2.1.1

The following table lists the dependencies for version 2.1.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.2.0 <2.6.0	Soft

2.1.0 and 2.0.2

The following table lists the dependencies for versions 2.1.0 and 2.0.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.2.0 <2.5.0	Soft

2.0.1

The following table lists the dependencies for version 2.0.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.2.0 <2.4.0	Soft

2.0.0

The following table lists the dependencies for version 2.0.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.2.0 <2.3.0	Soft

For more information about component dependencies, see the component recipe reference.

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

2.2.x

`defaultPort`

(Optional) The MQTT broker port to report when this component detects IP addresses. You must specify this parameter if you configure the MQTT broker to use a different port than the default port 8883.

Default: 8883

`includeIPv4LoopbackAddrs`

(Optional) You can enable this option to detect and report IPv4 loopback addresses. These are IP addresses, such as `localhost`, where a device can communicate with itself. Use this option in test environments where the core device and client device run on the same system.

Default: `false`

`includeIPv4LinkLocalAddrs`

(Optional) You can enable this option to detect and report IPv4 link-local addresses. Use this option if the core device's network doesn't have Dynamic Host Configuration Protocol (DHCP) or statically assigned IP addresses.

Default: `false`

`includeIPv6LoopbackAddrs`

(Optional) You can enable this option to detect and report IPv6 loopback addresses. These are IP addresses, such as `localhost`, where a device can communicate with itself. Use this option in test environments where the core device and client device run on the same system. You must set `includeIPv4Addrs` to `false` and `includeIPv6Addrs` to `true` to use this option.

Default: `false`

`includeIPv6LinkLocalAddrs`

(Optional) You can enable this option to detect and report IPv6 link-local addresses. Use this option if the core device's network doesn't have Dynamic Host Configuration Protocol (DHCP) or statically assigned IP addresses. You must set `includeIPv4Addrs` to `false` and `includeIPv6Addrs` to `true` to use this option.

Default: `false`

`includeIPv4Addrs`

(Optional) The default is set to `true`. You can enable this option to publish IPv4 addresses found on the core device.

Default: `true`

`includeIPv6Addrs`

(Optional) You can enable this option to publish IPv6 addresses found on the core device. Set `includeIPv4Addrs` to `false` to use this option.

Default: `false`

2.1.x

`defaultPort`

(Optional) The MQTT broker port to report when this component detects IP addresses. You must specify this parameter if you configure the MQTT broker to use a different port than the default port 8883.

Default: 8883

`includeIPv4LoopbackAddrs`

(Optional) You can enable this option to detect and report IPv4 loopback addresses. These are IP addresses, such as `localhost`, where a device can communicate with itself. Use this option in test environments where the core device and client device run on the same system.

Default: `false`

`includeIPv4LinkLocalAddrs`

(Optional) You can enable this option to detect and report IPv4 link-local addresses. Use this option if the core device's network doesn't have Dynamic Host Configuration Protocol (DHCP) or statically assigned IP addresses.

Default: `false`

2.0.x

`includeIPv4LoopbackAddrs`

(Optional) You can enable this option to detect and report IPv4 loopback addresses. These are IP addresses, such as `localhost`, where a device can communicate with itself. Use this option in test environments where the core device and client device run on the same system.

Default: `false`

`includeIPv4LinkLocalAddrs`

(Optional) You can enable this option to detect and report IPv4 link-local addresses. Use this option if the core device's network doesn't have Dynamic Host Configuration Protocol (DHCP) or statically assigned IP addresses.

Default: `false`

Local log file

This component uses the same log file as the Greengrass nucleus component.

Linux

```
/greengrass/v2/logs/greengrass.log
```

Windows

```
C:\greengrass\v2\logs\greengrass.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
  ```

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.2.5	Updates the component version for the Greengrass nucleus version 2.18.0 release.
2.2.4	Updates the component version for the Greengrass nucleus version 2.17.0 release.

Version	Changes
2.2.3	Version updated for Greengrass nucleus version 2.16.0 release.
2.2.2	Version updated for Greengrass nucleus version 2.15.0 release.
2.2.1	Version updated for Greengrass nucleus version 2.14.0 release.
2.2.0	Version updated for Greengrass nucleus version 2.13.0 release. New features • Adds support for IPv6. You can now use IPv6 for local messaging.
2.1.9	Bug fixes and improvements • Adjusts the IP acquired step to only send logs at the debug log level.
2.1.8	Version updated for Greengrass nucleus version 2.12.0 release.
2.1.7	Version updated for Greengrass nucleus version 2.11.0 release.
2.1.6	Version updated for Greengrass nucleus version 2.10.0 release.
2.1.5	Version updated for Greengrass nucleus version 2.9.0 release.
2.1.4	Version updated for Greengrass nucleus version 2.8.0 release.
2.1.3	Version updated for Greengrass nucleus version 2.7.0 release.
2.1.2	Bug fixes and improvements • Improves error messages that this component logs in certain scenarios. • Version updated for Greengrass nucleus version 2.6.0 release.
2.1.1	Version updated for Greengrass nucleus version 2.5.0 release.
2.1.0	Improvements • Adds the `defaultPort` parameter, which enables you to use a non-default MQTT broker port. • Updates to make log messages more clear.

Version	Changes
2.0.2	Version updated for Greengrass nucleus version 2.4.0 release.
2.0.1	Version updated for Greengrass nucleus version 2.3.0 release.
2.0.0	Initial version.

Firehose

The Firehose component (`aws.greengrass.KinesisFirehose`) publishes data through Amazon Data Firehose delivery streams to destinations, such as Amazon S3, Amazon Redshift, and Amazon OpenSearch Service. For more information, see What is Amazon Data Firehose? in the *Amazon Data Firehose Developer Guide*.

To publish to a Kinesis delivery stream with this component, publish a message to a topic where this component subscribes. By default, this component subscribes to the `kinesisfirehose/message` and `kinesisfirehose/message/binary/#` local publish/subscribe topics. You can specify other topics, including AWS IoT Core MQTT topics, when you deploy this component.

> **Note**
>
> This component provides similar functionality to the Firehose connector in AWS IoT Greengrass V1. For more information, see Firehose connector in the *AWS IoT Greengrass V1 Developer Guide*.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Input data
- Output data

- Local log file
- Licenses
- Changelog
- See also

Versions

This component has the following versions:

- 2.1.x
- 2.0.x

Type

This component is a Lambda component (`aws.greengrass.lambda`). The Greengrass nucleus runs this component's Lambda function using the Lambda launcher component.

For more information, see Component types.

Operating system

This component can be installed on Linux core devices only.

Requirements

This component has the following requirements:

- Your core device must meet the requirements to run Lambda functions. If you want the core device to run containerized Lambda functions, the device must meet the requirements to do so. For more information, see Lambda function requirements.
- Python version 3.7 installed on the core device and added to the PATH environment variable.
- The Greengrass device role must allow the `firehose:PutRecord` and `firehose:PutRecordBatch` actions, as shown in the following example IAM policy.

  JSON

```
{
  "Version":"2012-10-17",
```

```
  "Statement": [
    {
      "Action": [
        "firehose:PutRecord",
        "firehose:PutRecordBatch"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:firehose:us-east-1:123456789012:deliverystream/stream-name"
      ]
    }
  ]
}
```

You can dynamically override the default delivery stream in the input message payload for this component. If your application uses this feature, the IAM policy must include all target streams as resources. You can grant granular or conditional access to resources (for example, by using a wildcard * naming scheme).

- To receive output data from this component, you must merge the following configuration update for the legacy subscription router component (`aws.greengrass.LegacySubscriptionRouter`) when you deploy this component. This configuration specifies the topic where this component publishes responses.

  Legacy subscription router v2.1.x

  ```
  {
    "subscriptions": {
      "aws-greengrass-kinesisfirehose": {
        "id": "aws-greengrass-kinesisfirehose",
        "source": "component:aws.greengrass.KinesisFirehose",
        "subject": "kinesisfirehose/message/status",
        "target": "cloud"
      }
    }
  }
  ```

  Legacy subscription router v2.0.x

  ```
  {
    "subscriptions": {
      "aws-greengrass-kinesisfirehose": {
  ```

```
      "id": "aws-greengrass-kinesisfirehose",
      "source": "arn:aws:lambda:region:aws:function:aws-greengrass-
kinesisfirehose:version",
      "subject": "kinesisfirehose/message/status",
      "target": "cloud"
    }
  }
}
```

- Replace *region* with the AWS Region that you use.
- Replace *version* with the version of the Lambda function that this component runs. To find the Lambda function version, you must view the recipe for the version of this component that you want to deploy. Open this component's details page in the AWS IoT Greengrass console, and look for the **Lambda function** key-value pair. This key-value pair contains the name and version of the Lambda function.

> **Important**
>
> You must update the Lambda function version on the legacy subscription router every time you deploy this component. This ensures that you use the correct Lambda function version for the component version that you deploy.

For more information, see Create deployments.

- The Firehose component is supported to run in a VPC. To deploy this component in a VPC, the following is required.
  - The Firehose component must have connectivity to `firehose.region.amazonaws.com` which has the VPC endpoint of `com.amazonaws.region.kinesis-firehose`.

Endpoints and ports

This component must be able to perform outbound requests to the following endpoints and ports, in addition to endpoints and ports required for basic operation. For more information, see Allow device traffic through a proxy or firewall.

Endpoint	Port	Required	Description
`firehose. region.amazonaws.com`	443	Yes	Upload data to Firehose.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

2.1.10

The following table lists the dependencies for version 2.1.10 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.16.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.1.9

The following table lists the dependencies for version 2.1.9 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.15.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.1.8

The following table lists the dependencies for version 2.1.8 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.14.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.1.7

The following table lists the dependencies for version 2.1.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.13.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.1.6

The following table lists the dependencies for version 2.1.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.12.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.1.5

The following table lists the dependencies for version 2.1.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.11.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.1.4

The following table lists the dependencies for version 2.1.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.10.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft

Dependency	Compatible versions	Dependency type
Token exchange service	^2.0.0	Hard

2.1.3

The following table lists the dependencies for version 2.1.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.9.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.1.2

The following table lists the dependencies for version 2.1.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.8.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.1.1

The following table lists the dependencies for version 2.1.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.7.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.0.8 - 2.1.0

The following table lists the dependencies for versions 2.0.8 and 2.1.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.6.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.0.7

The following table lists the dependencies for version 2.0.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.5.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.0.6

The following table lists the dependencies for version 2.0.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.4.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.0.5

The following table lists the dependencies for version 2.0.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.3.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.0.4

The following table lists the dependencies for version 2.0.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.2.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft

Dependency	Compatible versions	Dependency type
Token exchange service	^2.0.0	Hard

2.0.3

The following table lists the dependencies for version 2.0.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.3 <2.1.0	Hard
Lambda launcher	>=1.0.0	Hard
Lambda runtimes	>=1.0.0	Soft
Token exchange service	>=1.0.0	Hard

For more information about component dependencies, see the component recipe reference.

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

Note

This component's default configuration includes Lambda function parameters. We recommend that you edit only the following parameters to configure this component on your devices.

`lambdaParams`

An object that contains the parameters for this component's Lambda function. This object contains the following information:

`EnvironmentVariables`

An object that contains the Lambda function's parameters. This object contains the following information:

`DEFAULT_DELIVERY_STREAM_ARN`

The ARN of the default Firehose delivery stream where the component sends data. You can override the destination stream with the `delivery_stream_arn` property in the input message payload.

> **Note**
>
> The core device role must allow the required actions on all target delivery streams. For more information, see Requirements.

`PUBLISH_INTERVAL`

(Optional) The maximum number of seconds to wait before the component publishes batched data to Firehose. To configure the component to publish metrics as it receives them, which means without batching, specify `0`.

This value can be at most 900 seconds.

Default: 10 seconds

`DELIVERY_STREAM_QUEUE_SIZE`

(Optional) The maximum number of records to retain in memory before the component rejects new records for the same delivery stream.

This value must be at least 2,000 records.

Default: 5,000 records

`containerMode`

(Optional) The containerization mode for this component. Choose from the following options:

- `NoContainer` – The component doesn't run in an isolated runtime environment.
- `GreengrassContainer` – The component runs in an isolated runtime environment inside the AWS IoT Greengrass container.

Default: `GreengrassContainer`

`containerParams`

(Optional) An object that contains the container parameters for this component. The component uses these parameters if you specify `GreengrassContainer` for `containerMode`.

This object contains the following information:

`memorySize`

(Optional) The amount of memory (in kilobytes) to allocate to the component.

Defaults to 64 MB (65,535 KB).

`pubsubTopics`

(Optional) An object that contains the topics where the component subscribes to receive messages. You can specify each topic and whether the component subscribes to MQTT topics from AWS IoT Core or local publish/subscribe topics.

This object contains the following information:

`0` – This is an array index as a string.

An object that contains the following information:

`type`

(Optional) The type of publish/subscribe messaging that this component uses to subscribe to messages. Choose from the following options:

- PUB_SUB – Subscribe to local publish/subscribe messages. If you choose this option, the topic can't contain MQTT wildcards. For more information about how to send messages from custom component when you specify this option, see Publish/subscribe local messages.
- `IOT_CORE` – Subscribe to AWS IoT Core MQTT messages. If you choose this option, the topic can contain MQTT wildcards. For more information about how to send messages from custom components when you specify this option, see Publish/subscribe AWS IoT Core MQTT messages.

Default: PUB_SUB

`topic`

(Optional) The topic to which the component subscribes to receive messages. If you specify `IotCore` for `type`, you can use MQTT wildcards (+ and #) in this topic.

Example: Configuration merge update (container mode)

```
{
  "lambdaExecutionParameters": {
    "EnvironmentVariables": {
      "DEFAULT_DELIVERY_STREAM_ARN": "arn:aws:firehose:us-
west-2:123456789012:deliverystream/mystream"
    }
  },
  "containerMode": "GreengrassContainer"
}
```

Example: Configuration merge update (no container mode)

```
{
  "lambdaExecutionParameters": {
    "EnvironmentVariables": {
      "DEFAULT_DELIVERY_STREAM_ARN": "arn:aws:firehose:us-
west-2:123456789012:deliverystream/mystream"
    }
  },
  "containerMode": "NoContainer"
}
```

Input data

This component accepts stream content on the following topics and sends the content to the target delivery stream. The component accepts two types of input data:

- JSON data on the `kinesisfirehose/message` topic.
- Binary data on the `kinesisfirehose/message/binary/#` topic.

Default topic for JSON data (local publish/subscribe): `kinesisfirehose/message`

The message accepts the following properties. Input messages must be in JSON format.

`request`

The data to send to the delivery stream and the target delivery stream, if different from the default stream.

Type: `object` that contains the following information:

`data`

The data to send to the delivery stream.

Type: `string`

`delivery_stream_arn`

(Optional) The ARN of the target Firehose delivery stream. Specify this property to override the default delivery stream.

Type: `string`

`id`

An arbitrary ID for the request. Use this property to map an input request to an output response. When you specify this property, the component sets the `id` property in the response object to this value.

Type: `string`

Example input

```
{
  "request": {
    "delivery_stream_arn": "arn:aws:firehose:region:account-id:deliverystream/
stream2-name",
    "data": "Data to send to the delivery stream."
  },
  "id": "request123"
}
```

Default topic for binary data (local publish/subscribe): `kinesisfirehose/message/binary/#`

Use this topic to send a message that contains binary data. The component doesn't parse binary data. The component streams the data as is.

To map the input request to an output response, replace the # wildcard in the message topic with an arbitrary request ID. For example, if you publish a message to `kinesisfirehose/message/binary/request123`, the `id` property in the response object is set to `request123`.

If you don't want to map a request to a response, you can publish your messages to `kinesisfirehose/message/binary/`. Be sure to include the trailing slash (/).

Output data

This component publishes responses as output data on the following MQTT topic by default. You must specify this topic as the `subject` in the configuration for the legacy subscription router component. For more information about how to subscribe to messages on this topic in your custom components, see Publish/subscribe AWS IoT Core MQTT messages.

Default topic (AWS IoT Core MQTT): `kinesisfirehose/message/status`

Example output

The response contains the status of each data record sent in the batch.

```
{
  "response": [
    {
      "ErrorCode": "error",
      "ErrorMessage": "test error",
      "id": "request123",
      "status": "fail"
    },
    {
      "firehose_record_id": "xyz2",
      "id": "request456",
      "status": "success"
    },
    {
      "firehose_record_id": "xyz3",
      "id": "request890",
      "status": "success"
    }
  ]
}
```

Note

If the component detects an error that can be retried, such as a connection error, it retries the publish in the next batch.

Local log file

This component uses the following log file.

```
/greengrass/v2/logs/aws.greengrass.KinesisFirehose.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* with the path to the AWS IoT Greengrass root folder.

  ```
  sudo tail -f /greengrass/v2/logs/aws.greengrass.KinesisFirehose.log
  ```

Licenses

This component includes the following third-party software/licensing:

- AWS SDK for Python (Boto3)/Apache License 2.0
- botocore/Apache License 2.0
- dateutil/PSF License
- docutils/BSD License, GNU General Public License (GPL), Python Software Foundation License, Public Domain
- jmespath/MIT License
- s3transfer/Apache License 2.0
- urllib3/MIT License

This component is released under the Greengrass Core Software License Agreement.

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.1.10	Version updated for Greengrass nucleus version 2.15.0 release.
2.1.9	Version updated for Greengrass nucleus version 2.14.0 release.

Version	Changes
2.1.8	Version updated for Greengrass nucleus version 2.13.0 release.
2.1.7	Version updated for Greengrass nucleus version 2.12.0 release.
2.1.6	Version updated for Greengrass nucleus version 2.11.0 release.
2.1.5	Version updated for Greengrass nucleus version 2.10.0 release.
2.1.4	Version updated for Greengrass nucleus version 2.9.0 release.
2.1.3	Version updated for Greengrass nucleus version 2.8.0 release.
2.1.2	Version updated for Greengrass nucleus version 2.7.0 release.
2.1.1	Version updated for Greengrass nucleus version 2.6.0 release.
2.1.0	New features • Adds support for HTTPS network proxy configurations. For more information, see Connect on port 443 or through a network proxy and Enable the core device to trust an HTTPS proxy.
2.0.8	Version updated for Greengrass nucleus version 2.5.0 release.
2.0.7	Version updated for Greengrass nucleus version 2.4.0 release.
2.0.6	Version updated for Greengrass nucleus version 2.3.0 release.
2.0.5	Version updated for Greengrass nucleus version 2.2.0 release.
2.0.4	Version updated for Greengrass nucleus version 2.1.0 release.
2.0.3	Initial version.

See also

- What is Amazon Data Firehose? in the *Amazon Data Firehose Developer Guide*

Lambda launcher

The Lambda launcher component (`aws.greengrass.LambdaLauncher`) starts and stops AWS Lambda functions on AWS IoT Greengrass core devices. This component also sets up any containerization and runs processes as the users that you specify.

> **Note**
>
> When you deploy a Lambda function component to a core device, the deployment also includes this component. For more information, see Run AWS Lambda functions.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Local log file
- Changelog

Versions

This component has the following versions:

- 2.0.x

Type

This component is a generic component (`aws.greengrass.generic`). The Greengrass nucleus runs the component's lifecycle scripts.

For more information, see Component types.

Operating system

This component can be installed on Linux core devices only.

Requirements

This component has the following requirements:

- Your core device must meet the requirements to run Lambda functions. If you want the core device to run containerized Lambda functions, the device must meet the requirements to do so. For more information, see Lambda function requirements.
- The Lambda launcher component is supported to run in a VPC.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

2.0.11 – 2.0.13

The following table lists the dependencies for versions 2.0.11 to 2.0.13 of this component.

Dependency	Compatible versions	Dependency type
Lambda manager	>=2.0.0 <2.4.0	Hard

2.0.9 – 2.0.10

The following table lists the dependencies for versions 2.0.9 to 2.0.10 of this component.

Dependency	Compatible versions	Dependency type
Lambda manager	>=2.0.0 <2.3.0	Hard

2.0.4 - 2.0.8

The following table lists the dependencies for versions 2.0.4 to 2.0.8 of this component.

Dependency	Compatible versions	Dependency type
Lambda manager	>=2.0.0 <2.2.0	Hard

2.0.3

The following table lists the dependencies for version 2.0.3 of this component.

Dependency	Compatible versions	Dependency type
Lambda manager	>=2.0.3 <2.1.0	Hard

For more information about component dependencies, see the component recipe reference.

Configuration

This component doesn't have any configuration parameters.

Local log file

This component uses the following log file.

```
/greengrass/v2/logs/lambdaFunctionComponentName.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* with the path to the AWS IoT Greengrass root folder, and replace *lambdaFunctionComponentName* with the name of the Lambda function component that this component launches.

  ```
  sudo tail -f /greengrass/v2/logs/lambdaFunctionComponentName.log
  ```

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.0.13	Bug fixes and improvements General bug fixes and improvements.
2.0.12	Bug fixes and improvements Fixes an issue where the Lambda launcher could throw an error if the previous process was not stopped properly.
2.0.11	Support for Lambda manager 2.3.0.
2.0.10	Bug fixes and improvements • General bug fixes and improvements.
2.0.9	Version updated for Greengrass nucleus version 2.5.0 release.
2.0.8	Version updated for Greengrass nucleus version 2.4.0 release.
2.0.7	Version updated for Greengrass nucleus version 2.3.0 release.
2.0.6	General performance improvements and bug fixes.
2.0.4	Bug fixes and improvements • Fixes an issue where the component doesn't correctly pass `AddGroupOwner` to the Lambda function container.
2.0.3	Initial version.

Lambda manager

The Lambda manager component (`aws.greengrass.LambdaManager`) manages work items and interprocess communication for AWS Lambda functions that run on the Greengrass core device.

> **Note**
>
> When you deploy a Lambda function component to a core device, the deployment also includes this component. For more information, see Run AWS Lambda functions.

Topics

- Versions
- Operating system
- Type
- Requirements
- Dependencies
- Configuration
- Local log file
- Changelog

Versions

This component has the following versions:

- 2.3.x
- 2.2.x
- 2.1.x
- 2.0.x

Operating system

This component can be installed on Linux core devices only.

Type

This component is a plugin component (`aws.greengrass.plugin`). The Greengrass nucleus runs this component in the same Java Virtual Machine (JVM) as the nucleus. The nucleus restarts when you change this component's version on the core device.

This component uses the same log file as the Greengrass nucleus. For more information, see Monitor AWS IoT Greengrass logs.

For more information, see Component types.

Requirements

This component has the following requirements:

- Your core device must meet the requirements to run Lambda functions. If you want the core device to run containerized Lambda functions, the device must meet the requirements to do so. For more information, see Lambda function requirements.
- The Lambda manager component is supported to run in a VPC.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

2.3.7

The following table lists the dependencies for version 2.3.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.17.0	Soft

2.3.6

The following table lists the dependencies for version 2.3.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.16.0	Soft

2.3.5

The following table lists the dependencies for version 2.3.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.15.0	Soft

2.3.4

The following table lists the dependencies for version 2.3.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.14.0	Soft

2.3.2 and 2.3.3

The following table lists the dependencies for version 2.3.2 and 2.3.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.13.0	Soft

2.2.10 and 2.3.1

The following table lists the dependencies for version 2.2.10 and 2.3.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.12.0	Soft

2.2.8 and 2.2.9

The following table lists the dependencies for version 2.2.8 and 2.2.9 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.11.0	Soft

2.2.7

The following table lists the dependencies for version 2.2.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.10.0	Soft

2.2.6

The following table lists the dependencies for version 2.2.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.9.0	Soft

2.2.5

The following table lists the dependencies for version 2.2.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.8.0	Soft

2.2.4

The following table lists the dependencies for version 2.2.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.7.0	Soft

2.2.1 - 2.2.3

The following table lists the dependencies for versions 2.2.1 to 2.2.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.6.0	Soft

2.2.0

The following table lists the dependencies for version 2.2.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.6.0	Soft

2.1.3 and 2.1.4

The following table lists the dependencies for versions 2.1.3 and 2.1.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.5.0	Soft

2.1.2

The following table lists the dependencies for version 2.1.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.4.0	Soft

2.1.1

The following table lists the dependencies for version 2.1.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.3.0	Soft

2.1.0

The following table lists the dependencies for version 2.1.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.2.0	Soft

2.0.x

The following table lists the dependencies for version 2.0.x of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.3 <2.1.0	Soft

For more information about component dependencies, see the component recipe reference.

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

`logHandlerMode`

> **Note**
>
> Only for lambda manager versions 2.3.0+

Used to choose the implementation of the Lambda log manager to use. Set the value to `optimized` to use fewer threads to read lambda logs.

`getResultTimeoutInSecond`

(Optional) The maximum amount of time in seconds that Lambda functions can run before they time out.

Default: `60`

Local log file

This component uses the same log file as the Greengrass nucleus component.

```
/greengrass/v2/logs/greengrass.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* with the path to the AWS IoT Greengrass root folder.

  ```
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.3.9	Updates the component version for the Greengrass nucleus version 2.18.0 release.
2.3.8	Updates the component version for the Greengrass nucleus version 2.17.0 release.
2.3.7	Version updated for Greengrass nucleus version 2.16.0 release.
2.3.6	Version updated for Greengrass nucleus version 2.15.0 release.

Version	Changes
2.3.5	Bug fixes and improvements • Improves performance by using epoll instead of nio when available.
2.3.4	Version updated for Greengrass nucleus version 2.13.0 release.
2.3.3	Bug fixes and improvements • General bug fixes and improvements.
2.3.2	Version updated for Greengrass nucleus version 2.12.0 release.
2.3.1	Bug fixes and improvements • Adjusts log levels for certain errors.
2.3.0	New features • Log handler was optimized to reduce CPU load. Use this feature by setting the configuration option `logHandlerMode` to `optimized`. Bug fixes and improvements • No longer logs the full stacktrace for `WorkQueueFullException`, improving logs and performance. • Sets lambda shutdown timeout from 15 seconds to 300 seconds in order to prevent shutdown timeouts. • Fixes an issue where on-demand lambdas may fail to restart after changing configuration.
2.2.11	Bug fixes and improvements • Fixes an issue where the LegacySubscriptionRouter configuration does not update when the Lambda configuration changes.
2.2.10	Version updated for Greengrass nucleus version 2.11.0 release.
2.2.9	Bug fixes and improvements Fixes an issue where the port number is corrupted due to a skewed clock.
2.2.8	Version updated for Greengrass nucleus version 2.10.0 release.

Version	Changes
2.2.7	Version updated for Greengrass nucleus version 2.9.0 release.
2.2.6	Version updated for Greengrass nucleus version 2.8.0 release.
2.2.5	New features • Adds support for MQTT topic wildcards in event sources where you subscribe to local publish/subscribe messages. This feature requires v2.6.0 or later of the Greengrass nucleus component. • Version updated for Greengrass nucleus version 2.7.0 release.
2.2.4	Version updated for Greengrass nucleus version 2.6.0 release.
2.2.3	Bug fixes and improvements • Fixes an issue where multiple instances of a Lambda function share a single cgroup. This component uses cgroups to manage resource usage for Lambda functions.
2.2.2	Bug fixes and improvements • Fixes an issue where pinned Lambda function components restart unexpectedly in certain scenarios.
2.2.1	Bug fixes and improvements • Changes this component's Greengrass nucleus dependency version constraints to fix a dependency resolution issue.
2.2.0	Bug fixes and improvements • Fixes an issue where Lambda functions couldn't write logs after a restart. • Fixes an issue where the legacy subscription router sends duplicate messages when there are wildcards in the topic. • Fixes an issue where non-pinned Lambda functions couldn't use the Greengrass interprocess communication (IPC) library in the AWS IoT Device SDK.

Version	Changes
2.1.4	Bug fixes and improvements • Fixes an issue that caused Lambda functions that use NodeJS runtimes to process only one message. • Version updated for Greengrass nucleus version 2.5.0 release.
2.1.3	Version updated for Greengrass nucleus version 2.4.0 release.
2.1.2	Version updated for Greengrass nucleus version 2.3.0 release.
2.1.1	Version updated for Greengrass nucleus version 2.2.0 release.
2.1.0	Version updated for Greengrass nucleus version 2.1.0 release.
2.0.3	Initial version.

Lambda runtimes

The Lambda runtimes component (`aws.greengrass.LambdaRuntimes`) provides the runtimes that Greengrass core devices use to run AWS Lambda functions.

> **Note**
>
> When you deploy a Lambda function component to a core device, the deployment also includes this component. For more information, see Run AWS Lambda functions.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Local log file

- Changelog

Versions

This component has the following versions:

- 2.0.x

Type

This component is a generic component (`aws.greengrass.generic`). The Greengrass nucleus runs the component's lifecycle scripts.

For more information, see Component types.

Operating system

This component can be installed on Linux core devices only.

Requirements

This component has the following requirements:

- Your core device must meet the requirements to run Lambda functions. If you want the core device to run containerized Lambda functions, the device must meet the requirements to do so. For more information, see Lambda function requirements.
- The Lambda runtimes component is supported to run in a VPC.

Dependencies

This component doesn't have any dependencies.

Configuration

This component doesn't have any configuration parameters.

Local log file

This component doesn't output logs.

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.0.9	Bug fixes and improvements Fixes an syntax warning with Python 3.12
2.0.8	Version updated for Greengrass nucleus version 2.5.0 release.
2.0.7	Version updated for Greengrass nucleus version 2.4.0 release.
2.0.6	Version updated for Greengrass nucleus version 2.3.0 release.
2.0.5	Version updated for Greengrass nucleus version 2.2.0 release.
2.0.4	Version updated for Greengrass nucleus version 2.1.0 release.
2.0.3	Initial version.

Legacy subscription router

The legacy subscription router (`aws.greengrass.LegacySubscriptionRouter`) manages subscriptions on the Greengrass core device. Subscriptions are a feature of AWS IoT Greengrass V1 that define the topics that Lambda functions can use for MQTT messaging on a core device. For more information, see Managed subscriptions in the MQTT messaging workflow in the *AWS IoT Greengrass V1 Developer Guide*.

You can use this component to enable subscriptions for connector components and Lambda function components that use the AWS IoT Greengrass Core SDK.

> **Note**
>
> The legacy subscription router component is required only if your Lambda function uses the `publish()` function in the AWS IoT Greengrass Core SDK. If you update your Lambda function code to use the interprocess communication (IPC) interface in the AWS IoT Device SDK V2, you don't need to deploy the legacy subscription router component. For more information, see the following interprocess communication services:

- Publish/subscribe local messages
- Publish/subscribe AWS IoT Core MQTT messages

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Local log file
- Changelog

Versions

This component has the following versions:

- 2.1.x
- 2.0.x

Type

This component is a generic component (`aws.greengrass.generic`). The Greengrass nucleus runs the component's lifecycle scripts.

For more information, see Component types.

Operating system

This component can be installed on Linux core devices only.

Requirements

This component has the following requirements:

- The legacy subscription router is supported to run in a VPC.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

2.1.15

The following table lists the dependencies for version 2.1.15 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.17.0	Soft

2.1.14

The following table lists the dependencies for version 2.1.14 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.16.0	Soft

2.1.13

The following table lists the dependencies for version 2.1.13 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.15.0	Soft

2.1.12

The following table lists the dependencies for version 2.1.12 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.14.0	Soft

2.1.11

The following table lists the dependencies for version 2.1.11 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.13.0	Soft

2.1.10

The following table lists the dependencies for version 2.1.10 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.12.0	Soft

2.1.9

The following table lists the dependencies for version 2.1.9 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.11.0	Soft

2.1.8

The following table lists the dependencies for version 2.1.8 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.10.0	Soft

2.1.7

The following table lists the dependencies for version 2.1.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.9.0	Soft

2.1.6

The following table lists the dependencies for version 2.1.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.8.0	Soft

2.1.5

The following table lists the dependencies for version 2.1.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.7.0	Soft

2.1.4

The following table lists the dependencies for version 2.1.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.6.0	Soft

2.1.3

The following table lists the dependencies for version 2.1.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.5.0	Soft

2.1.2

The following table lists the dependencies for version 2.1.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.4.0	Soft

2.1.1

The following table lists the dependencies for version 2.1.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.3.0	Soft

2.1.0

The following table lists the dependencies for version 2.1.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.2.0	Soft

2.0.3

The following table lists the dependencies for version 2.0.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.3 <2.1.0	Soft

For more information about component dependencies, see the component recipe reference.

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

v2.1.x

`subscriptions`

(Optional) The subscriptions to enable on the core device. This is an object, where each key is a unique ID, and each value is an object that defines the subscription for that connector. You must configure a subscription when you deploy a V1 connector component or a Lambda function that uses the AWS IoT Greengrass Core SDK.

Each subscription object contains the following information:

`id`

The unique ID of this subscription. This ID must match the key for this subscription object.

`source`

The Lambda function that uses the AWS IoT Greengrass Core SDK to publish MQTT messages on the topics that you specify in `subject`. Specify one of the following:

- The name of a Lambda function component on the core device.
  Specify the component name with the `component:` prefix, such as **`component:com.example.HelloWorldLambda`**.
- The Amazon Resource Name (ARN) of a Lambda function on the core device.

> **Important**
>
> If the version of the Lambda function changes, you must configure the subscription with the new version of the function. Otherwise, this component won't route the messages until the version matches the subscription.
> You must specify an Amazon Resource Name (ARN) that includes the version of the function to import. You can't use version aliases like `$LATEST`.

To deploy a subscription for a V1 connector component, specify the name of the component or the ARN of the connector component's Lambda function.

`subject`

The MQTT topic or topic filter on which the source and target can publish and receive messages. This value supports the + and # topic wildcards.

`target`

The target that receives the MQTT messages on the topics that you specify in `subject`. The subscription specifies that the `source` function publishes MQTT messages to AWS IoT Core or to a Lambda function on the core device. Specify one of the following:

- `cloud`. The `source` function publishes MQTT messages to AWS IoT Core.
- The name of a Lambda function component on the core device.
  Specify the component name with the `component:` prefix, such as **`component:com.example.HelloWorldLambda`**.
- The Amazon Resource Name (ARN) of a Lambda function on the core device.

> **Important**
>
> If the version of the Lambda function changes, you must configure the subscription with the new version of the function. Otherwise, this component won't route the messages until the version matches the subscription.
> You must specify an Amazon Resource Name (ARN) that includes the version of the function to import. You can't use version aliases like `$LATEST`.

Default: No subscriptions

Example configuration update (defining a subscription to AWS IoT Core)

The following example specifies that the `com.example.HelloWorldLambda` Lambda function component publishes MQTT message to AWS IoT Core on the `hello/world` topic.

```
{
  "subscriptions": {
    "Greengrass_HelloWorld_to_cloud": {
      "id": "Greengrass_HelloWorld_to_cloud",
      "source": "component:com.example.HelloWorldLambda",
      "subject": "hello/world",
      "target": "cloud"
    }
  }
}
```

Example configuration update (defining a subscription to another Lambda function)

The following example specifies that the `com.example.HelloWorldLambda` Lambda function component publishes MQTT messages to the `com.example.MessageRelay` Lambda function component on the `hello/world` topic.

```
{
  "subscriptions": {
    "Greengrass_HelloWorld_to_MessageRelay": {
      "id": "Greengrass_HelloWorld_to_MessageRelay",
      "source": "component:com.example.HelloWorldLambda",
      "subject": "hello/world",
      "target": "component:com.example.MessageRelay"
    }
  }
}
```

v2.0.x

`subscriptions`

(Optional) The subscriptions to enable on the core device. This is an object, where each key is a unique ID, and each value is an object that defines the subscription for that connector. You must configure a subscription when you deploy a V1 connector component or a Lambda function that uses the AWS IoT Greengrass Core SDK.

Each subscription object contains the following information:

`id`

The unique ID of this subscription. This ID must match the key for this subscription object.

`source`

The Lambda function that uses the AWS IoT Greengrass Core SDK to publish MQTT messages on the topics that you specify in `subject`. Specify the following:

- The Amazon Resource Name (ARN) of a Lambda function on the core device.

> **Important**
>
> If the version of the Lambda function changes, you must configure the subscription with the new version of the function. Otherwise, this component won't route the messages until the version matches the subscription.
> You must specify an Amazon Resource Name (ARN) that includes the version of the function to import. You can't use version aliases like `$LATEST`.

To deploy a subscription for a V1 connector component, specify the ARN of the connector component's Lambda function.

`subject`

The MQTT topic or topic filter on which the source and target can publish and receive messages. This value supports the + and # topic wildcards.

`target`

The target that receives the MQTT messages on the topics that you specify in `subject`. The subscription specifies that the `source` function publishes MQTT messages to AWS IoT Core or to a Lambda function on the core device. Specify one of the following:

- `cloud`. The `source` function publishes MQTT messages to AWS IoT Core.
- The Amazon Resource Name (ARN) of a Lambda function on the core device.

> **Important**
>
> If the version of the Lambda function changes, you must configure the subscription with the new version of the function. Otherwise, this component won't route the messages until the version matches the subscription.
> You must specify an Amazon Resource Name (ARN) that includes the version of the function to import. You can't use version aliases like `$LATEST`.

Default: No subscriptions

Example configuration update (defining a subscription to AWS IoT Core)

The following example specifies that the `Greengrass_HelloWorld` function publishes MQTT message to AWS IoT Core on the `hello/world` topic.

```
"subscriptions": {
  "Greengrass_HelloWorld_to_cloud": {
    "id": "Greengrass_HelloWorld_to_cloud",
    "source": "arn:aws:lambda:us-
west-2:123456789012:function:Greengrass_HelloWorld:5",
    "subject": "hello/world",
    "target": "cloud"
  }
}
```

Example configuration update (defining a subscription to another Lambda function)

The following example specifies that the `Greengrass_HelloWorld` function publishes MQTT messages to the `Greengrass_MessageRelay` on the `hello/world` topic.

```
"subscriptions": {
  "Greengrass_HelloWorld_to_MessageRelay": {
    "id": "Greengrass_HelloWorld_to_MessageRelay",
    "source": "arn:aws:lambda:us-
west-2:123456789012:function:Greengrass_HelloWorld:5",
    "subject": "hello/world",
    "target": "arn:aws:lambda:us-
west-2:123456789012:function:Greengrass_MessageRelay:5"
  }
```

```
}
```

Local log file

This component doesn't output logs.

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.1.17	Updates the component version for the Greengrass nucleus version 2.18.0 release.
2.1.16	Updates the component version for the Greengrass nucleus version 2.17.0 release.
2.1.15	Version updated for Greengrass nucleus version 2.16.0 release.
2.1.14	Version updated for Greengrass nucleus version 2.15.0 release.
2.1.13	Version updated for Greengrass nucleus version 2.14.0 release.
2.1.12	Version updated for Greengrass nucleus version 2.13.0 release.
2.1.11	Version updated for Greengrass nucleus version 2.12.0 release.
2.1.10	Version updated for Greengrass nucleus version 2.11.0 release.
2.1.9	Version updated for Greengrass nucleus version 2.10.0 release.
2.1.8	Version updated for Greengrass nucleus version 2.9.0 release.
2.1.7	Version updated for Greengrass nucleus version 2.8.0 release.
2.1.6	Version updated for Greengrass nucleus version 2.7.0 release.
2.1.5	Version updated for Greengrass nucleus version 2.6.0 release.
2.1.4	Version updated for Greengrass nucleus version 2.5.0 release.

Version	Changes
2.1.3	Version updated for Greengrass nucleus version 2.4.0 release.
2.1.2	Version updated for Greengrass nucleus version 2.3.0 release.
2.1.1	Version updated for Greengrass nucleus version 2.2.0 release.
2.1.0	Bug fixes and improvements • Adds support to specify component names instead of ARNs for `source` and `target`. If you specify a component name for a subscription, you don't need to reconfigure the subscription each time the version of the Lambda function changes.
2.0.3	Initial version.

Local debug console

The local debug console component (`aws.greengrass.LocalDebugConsole`) provides a local dashboard that displays information about your AWS IoT Greengrass core devices and its components. You can use this dashboard to debug your core device and manage local components.

> **Important**
>
> We recommend that you use this component in only development environments, not production environments. This component provides access to information and operations that you typically won't need in a production environment. Follow the principle of least privilege by deploying this component to only core devices where you need it.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies

- Configuration
- Usage
- Local log file
- Changelog

Versions

This component has the following versions:

- 2.4.x
- 2.3.x
- 2.2.x
- 2.1.x
- 2.0.x

Type

This component is a plugin component (`aws.greengrass.plugin`). The Greengrass nucleus runs this component in the same Java Virtual Machine (JVM) as the nucleus. The nucleus restarts when you change this component's version on the core device.

This component uses the same log file as the Greengrass nucleus. For more information, see Monitor AWS IoT Greengrass logs.

For more information, see Component types.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

Requirements

This component has the following requirements:

- You use a user name and password to sign in to the dashboard. The username, which is debug, is provided for you. You must use the AWS IoT Greengrass CLI to create a temporary password that authenticates you with the dashboard on a core device. You must be able to use the AWS IoT Greengrass CLI to use the local debug console. For more information, see the Greengrass CLI requirements. For more information about how to generate the password and sign in, see Local debug console component usage.
- The local debug console component is supported to run in a VPC.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

2.4.6

The following table lists the dependencies for version 2.4.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.10.0 <2.17.0	Hard
Greengrass CLI	>=2.10.0 <2.17.0	Hard

2.4.5

The following table lists the dependencies for version 2.4.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.10.0 <2.16.0	Hard
Greengrass CLI	>=2.10.0 <2.16.0	Hard

2.4.4

The following table lists the dependencies for version 2.4.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.10.0 <2.15.0	Hard
Greengrass CLI	>=2.10.0 <2.15.0	Hard

2.4.3

The following table lists the dependencies for version 2.4.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.10.0 <2.14.0	Hard
Greengrass CLI	>=2.10.0 <2.14.0	Hard

2.4.1 – 2.4.2

The following table lists the dependencies for versions 2.4.1 to 2.4.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.10.0 <2.13.0	Hard
Greengrass CLI	>=2.10.0 <2.13.0	Hard

2.4.0

The following table lists the dependencies for version 2.4.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.10.0 <2.12.0	Hard

Dependency	Compatible versions	Dependency type
Greengrass CLI	>=2.10.0 <2.12.0	Hard

2.3.0 and 2.3.1

The following table lists the dependencies for version 2.3.0 and 2.3.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.10.0 <2.12.0	Hard
Greengrass CLI	>=2.10.0 <2.12.0	Hard

2.2.9

The following table lists the dependencies for version 2.2.9 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.12.0	Hard
Greengrass CLI	>=2.1.0 <2.12.0	Hard

2.2.8

The following table lists the dependencies for version 2.2.8 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.11.0	Hard
Greengrass CLI	>=2.1.0 <2.11.0	Hard

2.2.7

The following table lists the dependencies for version 2.2.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.10.0	Hard
Greengrass CLI	>=2.1.0 <2.10.0	Hard

2.2.6

The following table lists the dependencies for version 2.2.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.9.0	Hard
Greengrass CLI	>=2.1.0 <2.9.0	Hard

2.2.5

The following table lists the dependencies for version 2.2.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.8.0	Hard
Greengrass CLI	>=2.1.0 <2.8.0	Hard

2.2.4

The following table lists the dependencies for version 2.2.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.7.0	Hard
Greengrass CLI	>=2.1.0 <2.7.0	Hard

2.2.3

The following table lists the dependencies for version 2.2.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.6.0	Hard
Greengrass CLI	>=2.1.0 <2.6.0	Hard

2.2.2

The following table lists the dependencies for version 2.2.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.5.0	Hard
Greengrass CLI	>=2.1.0 <2.5.0	Hard

2.2.1

The following table lists the dependencies for version 2.2.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.4.0	Hard
Greengrass CLI	>=2.1.0 <2.4.0	Hard

2.2.0

The following table lists the dependencies for version 2.2.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.3.0	Hard

Dependency	Compatible versions	Dependency type
Greengrass CLI	>=2.1.0 <2.3.0	Hard

2.1.0

The following table lists the dependencies for version 2.1.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.2.0	Hard
Greengrass CLI	>=2.1.0 <2.2.0	Hard

2.0.x

The following table lists the dependencies for version 2.0.x of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.3 <2.1.0	Soft
Greengrass CLI	>=2.0.3 <2.1.0	Soft

For more information about component dependencies, see the component recipe reference.

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

v2.1.x - v2.4.x

`httpsEnabled`

(Optional) You can enable HTTPS communication for the local debug console. If you enable HTTPS communication, the local debug console creates a self-signed certificate. Web browsers show security warnings for websites that use self-signed certificates, so you must

manually verify the certificate. Then, you can bypass the warning. For more information, see Usage.

Default: `true`

`port`

(Optional) The port at which to provide the local debug console.

Default: `1441`

`websocketPort`

(Optional) The websocket port to use for the local debug console.

Default: `1442`

`bindHostname`

(Optional) The hostname to use for the local debug console.

If you run the AWS IoT Greengrass Core software in a Docker container, set this parameter to `0.0.0.0`, so you can open the local debug console outside the Docker container.

Default: `localhost`

Example: Configuration merge update

The following example configuration specifies to open the local debug console on non-default ports and disable HTTPS.

```
{
  "httpsEnabled": false,
  "port": "10441",
  "websocketPort": "10442"
}
```

v2.0.x

`port`

(Optional) The port at which to provide the local debug console.

Default: `1441`

`websocketPort`

(Optional) The websocket port to use for the local debug console.

Default: `1442`

`bindHostname`

(Optional) The hostname to use for the local debug console.

If you run the AWS IoT Greengrass Core software in a Docker container, set this parameter to `0.0.0.0`, so you can open the local debug console outside the Docker container.

Default: `localhost`

Example: Configuration merge update

The following example configuration specifies to open the local debug console on non-default ports.

```
{
  "port": "10441",
  "websocketPort": "10442"
}
```

Usage

To use the local debug console, create a session from the Greengrass CLI. When you create a session, the Greengrass CLI provides a user name and temporary password that you can use to sign in to the local debug console.

Follow these instructions to open the local debug console on your core device or on your development computer.

v2.1.x - v2.4.x

In versions 2.1.0 and later, the local debug console uses HTTPS by default. When HTTPS is enabled, the local debug console creates a self-signed certificate to secure the connection. Your web browser shows a security warning when you open the local debug console because of this self-signed certificate. When you create a session with the Greengrass CLI, the output

includes the certificate's fingerprints, so you can verify that the certificate is legitimate and the connection is secure.

You can disable HTTPS. For more information, see Local debug console configuration.

To open the local debug console

1. (Optional) To view the local debug console on your development computer, you can forward the console's port over SSH. However, you must first enable the `AllowTcpForwarding` option in your core device's SSH configuration file. This option is enabled by default. Run the following command on your development computer to view the dashboard at `localhost:1441` on your development computer.

   ```
   ssh -L 1441:localhost:1441 -L 1442:localhost:1442 username@core-device-ip-address
   ```

   > **Note**
   >
   > You can change the default ports from 1441 and 1442. For more information, see Local debug console configuration.

2. Create a session to use the local debug console. When you create a session, you generate a password that you use to authenticate. The local debug console requires a password to increase security, because you can use this component to view important information and perform operations on the core device. The local debug console also creates a certificate to secure the connection if you enable HTTPS in the component configuration. HTTPS is enabled by default.

   Use the AWS IoT Greengrass CLI to create the session. This command generates a random 43-character password that expires after 8 hours. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass V2 root folder.

   Linux or Unix

   ```
   sudo /greengrass/v2/bin/greengrass-cli get-debug-password
   ```

Windows

```
C:\greengrass\v2\bin\greengrass-cli get-debug-password
```

The command output looks like the following example if you have configured the local debug console to use HTTPS. You use the certificate fingerprints to verify that the connection is secure when you open the local debug console.

```
Username: debug
Password: bEDp3MOHdj8ou2w5de_sCBI2XAaguy3a8XxREXAMPLE
Password expires at: 2021-04-01T17:01:43.921999931-07:00
The local debug console is configured to use TLS security. The certificate is
 self-signed so you will need to bypass your web browser's security warnings to
 open the console.
Before you bypass the security warning, verify that the certificate fingerprint
 matches the following fingerprints.
SHA-256: 15 0B 2C E2 54 8B 22 DE 08 46 54 8A B1 2B 25 DE FB 02 7D 01 4E 4A 56 67
 96 DA A6 CC B1 D2 C4 1B
SHA-1: BC 3E 16 04 D3 80 70 DA E0 47 25 F9 90 FA D6 02 80 3E B5 C1
```

The debug view component creates a session that lasts for 8 hours. After that, you must generate a new password to view the local debug console again.

3. Open and sign in to the dashboard. View the dashboard on your Greengrass core device, or on your development computer if you forward the port over SSH. Do one of the following:

   - If you enabled HTTPS in the local debug console, which is the default setting, do the following:

     a. Open `https://localhost:1441` on your core device, or on your development computer if you forwarded the port over SSH.

        Your browser might show a security warning about an invalid security certificate.

     b. If your browser shows a security warning, verify the certificate is legitimate and bypass the security warning. Do the following:

        i. Find the SHA-256 or SHA-1 fingerprint for the certificate, and verify that it matches the SHA-256 or SHA-1 fingerprint that the `get-debug-password` command previously printed. Your browser might provide one or both

fingerprints. Consult your browser's documentation to view the certificate and its fingerprints. In some browsers, the certificate fingerprint is called a thumbprint.

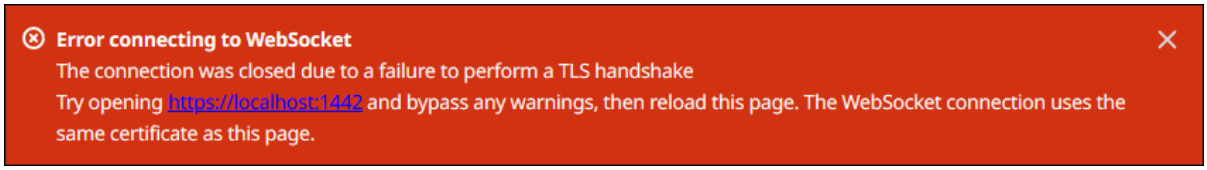
> **Note**
>
> If the certificate fingerprint doesn't match, go to Step 2 to create a new session. If the certificate fingerprint still doesn't match, your connection might be insecure.

ii. If the certificate fingerprint matches, bypass your browser's security warning to open the local debug console. Consult your browser's documentation to bypass the browser security warning.

c. Sign in to the website using the user name and password that the `get-debug-password` command printed earlier.

The local debug console opens.

d. If the local debug console shows an error that says it can't connect to the WebSocket due to a failed TLS handshake, you must bypass the self-signed security warning for the WebSocket URL.



Do the following:

i. Open `https://localhost:1442` in the same browser where you opened the local debug console.

ii. Verify the certificate and bypass the security warning.

Your browser might show an HTTP 404 page after you bypass the warning.

iii. Open `https://localhost:1441` again.

The local debug console shows information about the core device.

- If you disabled HTTPS in the local debug console, do the following:

a. Open `http://localhost:1441` on your core device, or open it on your development computer if you forwarded the port over SSH.

b. Sign in to the website using the user name and password that the `get-debug-password` command previously printed.

   The local debug console opens.

v2.0.x

To open the local debug console

1. (Optional) To view the local debug console on your development computer, you can forward the console's port over SSH. However, you must first enable the `AllowTcpForwarding` option in your core device's SSH configuration file. This option is enabled by default. Run the following command on your development computer to view the dashboard at `localhost:1441` on your development computer.

   ```
   ssh -L 1441:localhost:1441 -L 1442:localhost:1442 username@core-device-ip-address
   ```

   > **Note**
   >
   > You can change the default ports from 1441 and 1442. For more information, see Local debug console configuration.

2. Create a session to use the local debug console. When you create a session, you generate a password that you use to authenticate. The local debug console requires a password to increase security, because you can use this component to view important information and perform operations on the core device.

   Use the AWS IoT Greengrass CLI to create the session. This command generates a random 43-character password that expires after 8 hours. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass V2 root folder.

   Linux or Unix

   ```
   sudo /greengrass/v2/bin/greengrass-cli get-debug-password
   ```

Windows

```
C:\greengrass\v2\bin\greengrass-cli get-debug-password
```

The command output looks like the following example.

```
Username: debug
Password: bEDp3MOHdj8ou2w5de_sCBI2XAaguy3a8XxREXAMPLE
Password will expire at: 2021-04-01T17:01:43.921999931-07:00
```

The debug view component creates a session lasts for 4 hours, and then you must generate a new password to view the local debug console again.

3. Open `http://localhost:1441` on your core device, or open it on your development computer if you forwarded the port over SSH.
4. Sign in to the website using the user name and password that the `get-debug-password` command previously printed.

   The local debug console opens.

Local log file

This component uses the same log file as the Greengrass nucleus component.

Linux

```
/greengrass/v2/logs/greengrass.log
```

Windows

```
C:\greengrass\v2\logs\greengrass.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

Linux

```
sudo tail -f /greengrass/v2/logs/greengrass.log
```

Windows (PowerShell)

```
Get-Content C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
```

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.4.8	Updates the component version for the Greengrass nucleus version 2.18.0 release.
2.4.7	Updates the component version for the Greengrass nucleus version 2.17.0 release.
2.4.6	Version updated for Greengrass nucleus version 2.16.0 release.
2.4.5	Version updated for Greengrass nucleus version 2.15.0 release.
2.4.4	Version updated for Greengrass nucleus version 2.14.0 release.
2.4.3	Version updated for Greengrass nucleus version 2.13.0 release. Bug fixes and improvements • Fixes an issue that incorrectly displays `STREAM_MANAGER_EXPORTER_MAX_BANDWIDTH` in megabits per second (Mbps) instead of bytes per second (Bps).
2.4.2	Bug fixes and improvements • General bug fixes and improvements.
2.4.1	Version updated for Greengrass nucleus version 2.12.0 release.

Version	Changes
2.4.0	New features • Adds stream manager debugging console.
2.3.1	Version updated for Greengrass nucleus version 2.11.0 release.
2.3.0	Version updated for Greengrass nucleus version 2.10.0 release. New features • Includes PubSub and AWS IoT Core MQTT debug client.
2.2.7	Version updated for Greengrass nucleus version 2.9.0 release.
2.2.6	Version updated for Greengrass nucleus version 2.8.0 release.
2.2.5	Version updated for Greengrass nucleus version 2.7.0 release.
2.2.4	Version updated for Greengrass nucleus version 2.6.0 release.
2.2.3	Bug fixes and improvements • Fixes an issue that prevented startup when the component couldn't decrypt the keystore that holds the SSL private key. • Version updated for Greengrass nucleus version 2.5.0 release.
2.2.2	Version updated for Greengrass nucleus version 2.4.0 release.
2.2.1	Version updated for Greengrass nucleus version 2.3.0 release.
2.2.0	Version updated for Greengrass nucleus version 2.2.0 release.
2.1.0	New features • Uses HTTPS to secure your connection to the local debug console. HTTPS is enabled by default. Bug fixes and improvements • You can dismiss flashbar messages in the configuration editor.
2.0.3	Initial version.

Log manager

> **Warning**
>
> We recommend upgrading to Log manager v2.3.5 or later. Version 2.3.5 optimizes Log manager configuration writes, reducing IO operations and improving log upload speed, overall device performance and possibly extending device life.

The Log manager component (`aws.greengrass.LogManager`) uploads logs from AWS IoT Greengrass core devices to Amazon CloudWatch Logs. You can upload logs from the Greengrass nucleus, other Greengrass components, and other applications and services that aren't Greengrass components. For more information about how to monitor logs in CloudWatch Logs and on the local file system, see Monitor AWS IoT Greengrass logs.

The following considerations apply when you use the log manager component to write to CloudWatch Logs:

- **Log delays**

  The Log manager component version 2.2.8 (and earlier) processes and uploads logs from only rotated log files. By default, the AWS IoT Greengrass Core software rotates log files every hour or after they are 1,024 KB. As a result, the Log manager component uploads logs only after the AWS IoT Greengrass Core software or a Greengrass component writes over 1,024 KB worth of logs. You can configure a lower log file size limit to cause log files to rotate more often. This causes the Log manager component to upload logs to CloudWatch Logs more frequently.

  The Log manager component version 2.3.0 (and later) processes and uploads all logs. When you write a new log, Log manager version 2.3.0 (and later) processes and directly uploads that active log file instead of waiting for it to be rotated. This means that you can view the new log in 5 minutes or less.

  The Log manager component uploads new logs periodically. By default, the Log manager component uploads new logs every 5 minutes. You can configure a lower upload interval, so the Log manager component uploads logs to CloudWatch Logs more frequently by configuring the `periodicUploadIntervalSec`. For more information about how to configure this periodic interval, see Configuration.

Logs can be uploaded in near real-time from the same Greengrass file system. If you need to observe logs in real time, consider using file system logs.

> **Note**
>
> If you're using different file systems to write logs to, Log manager reverts back to the behavior in Log manager component versions 2.2.8 and earlier. For information about accessing file system logs, see Access file system logs.

- **Clock skew**

  The Log manager component uses the standard Signature Version 4 signing process to create API requests to CloudWatch Logs. If the system time on a core device is out of sync by more than 15 minutes, then CloudWatch Logs rejects the requests. For more information, see Signature Version 4 signing process in the *AWS General Reference*.

For information about the log groups and log streams to which this component uploads logs, see Usage.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Usage
- Local log file
- Changelog

Versions

This component has the following versions:

- 2.3.x

- 2.2.x
- 2.1.x
- 2.0.x

Type

This component is a plugin component (`aws.greengrass.plugin`). The Greengrass nucleus runs this component in the same Java Virtual Machine (JVM) as the nucleus. The nucleus restarts when you change this component's version on the core device.

This component uses the same log file as the Greengrass nucleus. For more information, see Monitor AWS IoT Greengrass logs.

For more information, see Component types.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

Requirements

This component has the following requirements:

- The Greengrass device role must allow the `logs:CreateLogGroup`, `logs:CreateLogStream`, `logs:PutLogEvents`, and `logs:DescribeLogStreams` actions, as shown in the following example IAM policy.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
```

```
        "logs:DescribeLogStreams"
      ],
      "Effect": "Allow",
      "Resource": "arn:aws:logs:*:*:*"
    }
  ]
}
```

> **Note**
>
> The Greengrass device role that you create when you install the AWS IoT Greengrass Core software includes the permissions in this example policy by default.

For more information, see Using identity-based policies (IAM policies) for CloudWatch Logs in the *Amazon CloudWatch Logs User Guide*.

- The Log manager component is supported to run in a VPC. To deploy this component in a VPC, the following is required.
  - The Log manager component must have connectivity to `logs.region.amazonaws.com` which has the VPC endpoint of `com.amazonaws.us-east-1.logs`.

Endpoints and ports

This component must be able to perform outbound requests to the following endpoints and ports, in addition to endpoints and ports required for basic operation. For more information, see Allow device traffic through a proxy or firewall.

Endpoint	Port	Required	Description
`logs.region.amazonaws.com`	443	No	Required if you write logs to CloudWatch Logs.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

2.3.13 and 2.3.14

The following table lists the dependencies for versions 2.3.13 and 2.3.14 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.19.0	Soft

2.3.12

The following table lists the dependencies for version 2.3.12 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.18.0	Soft

2.3.11

The following table lists the dependencies for version 2.3.11 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.17.0	Soft

2.3.10

The following table lists the dependencies for version 2.3.10 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.16.0	Soft

2.3.9

The following table lists the dependencies for version 2.3.9 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.15.0	Soft

2.3.8

The following table lists the dependencies for version 2.3.8 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.14.0	Soft

2.3.7

The following table lists the dependencies for version 2.3.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.13.0	Soft

2.3.5 and 2.3.6

The following table lists the dependencies for versions 2.3.5 and 2.3.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.12.0	Soft

2.3.3 – 2.3.4

The following table lists the dependencies for versions 2.3.3 to 2.3.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.11.0	Soft

2.2.8 – 2.3.2

The following table lists the dependencies for versions 2.2.8 to 2.3.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.10.0	Soft

2.2.7

The following table lists the dependencies for version 2.2.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.9.0	Soft

2.2.6

The following table lists the dependencies for version 2.2.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.8.0	Soft

2.2.5

The following table lists the dependencies for version 2.2.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.7.0	Soft

2.2.1 - 2.2.4

The following table lists the dependencies for versions 2.2.1 - 2.2.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.6.0	Soft

2.1.3 and 2.2.0

The following table lists the dependencies for versions 2.1.3 and 2.2.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.5.0	Soft

2.1.2

The following table lists the dependencies for version 2.1.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.4.0	Soft

2.1.1

The following table lists the dependencies for version 2.1.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.3.0	Soft

2.1.0

The following table lists the dependencies for version 2.1.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.1.0 <2.2.0	Soft

2.0.x

The following table lists the dependencies for version 2.0.x of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.3 <2.1.0	Soft

For more information about component dependencies, see the component recipe reference.

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

v2.3.10

`logsUploaderConfiguration`

(Optional) The configuration for logs that the Log manager component uploads. This object contains the following information:

`systemLogsConfiguration`

(Optional) The configuration for AWS IoT Greengrass Core software system logs, which include logs from the Greengrass nucleus and plugin components. Specify this configuration to enable the Log manager component to manage system logs. This object contains the following information:

`uploadToCloudWatch`

(Optional) You can upload system logs to CloudWatch Logs.

Default: `false`

`minimumLogLevel`

(Optional) The minimum level of log messages to upload. This minimum level applies only if you configure the Greengrass nucleus component to output JSON format logs. To enable JSON format logs, specify `JSON` for the logging format parameter (`logging.format`).

Choose from the following log levels, listed here in level order:

- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`

Default: `INFO`

`diskSpaceLimit`

(Optional) The maximum total size of Greengrass system log files, in the unit you specify in `diskSpaceLimitUnit`. After the total size of Greengrass system log files exceeds this maximum total size, the AWS IoT Greengrass Core software deletes the oldest Greengrass system log files.

This parameter is equivalent to the log size limit parameter (`totalLogsSizeKB`) of the Greengrass nucleus component. The AWS IoT Greengrass Core software uses the minimum of the two values as the maximum total Greengrass system log size.

`diskSpaceLimitUnit`

(Optional) The unit for the `diskSpaceLimit`. Choose from the following options:

- `KB` – kilobytes
- `MB` – megabytes
- `GB` – gigabytes

Default: `KB`

`deleteLogFileAfterCloudUpload`

(Optional) You can delete a log file after the log manager component uploads the logs to CloudWatch Logs.

Default: `false`

`componentLogsConfigurationMap`

(Optional) A map of log configurations for components on the core device. Each `componentName` object in this map defines the log configuration for the component or application. The Log manager component uploads these component logs to CloudWatch Logs.

> **Important**
>
> We strongly recommend using a single configuration key per component. You should only target a group of files that have only one log file that's actively being written to when using the `logFileRegex`. Not following this recommendation may lead to duplicate logs getting uploaded to CloudWatch. If you are targeting multiple active log files with a single regex, we recommend you upgrade to Log manager v2.3.1 or later and consider changing your configuration using the example configuration.

> **Note**
>
> If you're upgrading from a version of Log manager earlier than v2.2.0, you can continue to use the `componentLogsConfiguration` list instead of `componentLogsConfigurationMap`. However, we strongly recommend that you use the map format so that you can use merge and reset updates to modify configurations for specific components. For information about the `componentLogsConfiguration` parameter, see the configuration parameters for v2.1.x of this component.

`componentName`

The log configuration for the *`componentName`* component or application for this log configuration. You can specify the name of a Greengrass component or another value to identify this log group.

Each object contains the following information:

`minimumLogLevel`

(Optional) The minimum level of log messages to upload. This minimum level applies only if this component's logs use a specific JSON format, which you can find in the AWS IoT Greengrass logging module repository on GitHub.

Choose from the following log levels, listed here in level order:

- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`

Default: `INFO`

`diskSpaceLimit`

(Optional) The maximum total size of all log files for this component, in the unit you specify in `diskSpaceLimitUnit`. After the total size of this component's log files exceeds this maximum total size, the AWS IoT Greengrass Core software deletes this component's oldest log files.

This parameter is related to the log size limit parameter (`totalLogsSizeKB`) of the Greengrass nucleus component. The AWS IoT Greengrass Core software uses the minimum of the two values as the maximum total log size for this component.

> **Note**
>
> The log manager component applies this limit only after it uploads a component's log files to CloudWatch Logs. For a component that writes its own log files, files that aren't uploaded (for example, while the device is offline or uploads fail) aren't deleted, so the total size can exceed this limit.

`diskSpaceLimitUnit`

(Optional) The unit for the `diskSpaceLimit`. Choose from the following options:

- `KB` – kilobytes
- `MB` – megabytes
- `GB` – gigabytes

Default: KB

`logFileDirectoryPath`

(Optional) The path to the folder that contains this component's log files.

You don't need to specify this parameter for Greengrass components that print to standard output (stdout) and standard error (stderr).

Default: */greengrass/v2*/logs.

`logFileRegex`

(Optional) A regular expression that specifies the log file name format that the component or application uses. The log manager component uses this regular expression to identify log files in the folder at `logFileDirectoryPath`.

You don't need to specify this parameter for Greengrass components that print to standard output (stdout) and standard error (stderr).

If your component or application rotates log files, specify a regex that matches the rotated log file names. For example, you might specify **`hello_world\\\\w*.log`** to upload logs for a Hello World application. The `\\\\w*` pattern matches zero or more word characters, which includes alphanumeric characters and underscores. This regex matches log files with and without timestamps in their name. In this example, the log manager uploads the following log files:

- `hello_world.log` – The most recent log file for the Hello World application.
- `hello_world_2020_12_15_17_0.log` – An older log file for the Hello World application.

Default: *componentName*`\\\\w*.log`, where *componentName* is the name of the component for this log configuration.

`deleteLogFileAfterCloudUpload`

(Optional) You can delete a log file after the log manager component uploads the logs to CloudWatch Logs.

Default: `false`

`multiLineStartPattern`

(Optional) A regular expression that identifies when a log message on a new line is a new log message. If the regular expression doesn't match the new line, the log

manager component appends the new line to the log message for the previous line.

By default, a line begins a new log message only if it starts with a `YYYY-MM-DD` date (optionally after `[` or `(` or `{`. Otherwise, the log manager component appends that line to the current log message. In v2.3.4 and earlier, any line that doesn't start with a whitespace character is a new log message.

`periodicUploadIntervalSec`

(Optional) The period in seconds at which the log manager component checks for new log files to upload.

Default: `300` (5 minutes)

Minimum: `0.000001` (1 microsecond)

`updateToTlogIntervalSec`

(Optional) The period in seconds for how often the nucleus writes Amazon CloudWatch Events log-upload event details to the local transaction log (`config.tlog`). Defaults to the value specified in `periodicUploadIntervalSec`. You can modify this parameter to increase the writing interval.

Default: `periodicUploadIntervalSec`

Minimum: `periodicUploadIntervalSec`

`deprecatedVersionSupport`

Indicates whether the Log manager should use logging speed improvements introduced in Log manager v2.3.5. Set the value to `false` to use the improvements.

If you set this value to `false` when you upgrade from Log manager v2.3.1 or earlier duplicate log entries may be uploaded.

The default is `true`.

Example: Configuration merge update

The following example configuration specifies to upload system logs and `com.example.HelloWorld` component logs to CloudWatch Logs.

```
{
  "logsUploaderConfiguration": {
    "systemLogsConfiguration": {
      "uploadToCloudWatch": "true",
      "minimumLogLevel": "INFO",
      "diskSpaceLimit": "10",
      "diskSpaceLimitUnit": "MB",
      "deleteLogFileAfterCloudUpload": "false"
    },
    "componentLogsConfigurationMap": {
      "com.example.HelloWorld": {
        "minimumLogLevel": "INFO",
        "diskSpaceLimit": "20",
        "diskSpaceLimitUnit": "MB",
        "deleteLogFileAfterCloudUpload": "false"
      }
    }
  },
  "periodicUploadIntervalSec": "300",
  "deprecatedVersionSupport": "false"
}
```

Example: Configuration to upload multiple active log files using Log manager v2.3.1

The following example configuration is the recommended example if you want to target multiple active log files. This example configuration specifies what active log files you want to upload to CloudWatch. Using this configuration example configuration will also upload any rotated files that match the `logFileRegex`. This example configuration is supported on Log manager v2.3.1.

```
{
  "logsUploaderConfiguration": {
    "componentLogsConfigurationMap": {
      "com.example.A": {
        "logFileRegex": "com.example.A\\w*.log",
        "deleteLogFileAfterCloudUpload": "false"
      }
      "com.example.B": {
        "logFileRegex": "com.example.B\\w*.log",
        "deleteLogFileAfterCloudUpload": "false"
      }
    }
```

```
  },
  "periodicUploadIntervalSec": "10"
}
```

v2.3.6 – v2.3.9

`logsUploaderConfiguration`

(Optional) The configuration for logs that the Log manager component uploads. This object contains the following information:

`systemLogsConfiguration`

(Optional) The configuration for AWS IoT Greengrass Core software system logs, which include logs from the Greengrass nucleus and plugin components. Specify this configuration to enable the Log manager component to manage system logs. This object contains the following information:

`uploadToCloudWatch`

(Optional) You can upload system logs to CloudWatch Logs.

Default: `false`

`minimumLogLevel`

(Optional) The minimum level of log messages to upload. This minimum level applies only if you configure the Greengrass nucleus component to output JSON format logs. To enable JSON format logs, specify `JSON` for the logging format parameter (`logging.format`).

Choose from the following log levels, listed here in level order:

- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`

Default: `INFO`

`diskSpaceLimit`

(Optional) The maximum total size of Greengrass system log files, in the unit you specify in `diskSpaceLimitUnit`. After the total size of Greengrass system log files

exceeds this maximum total size, the AWS IoT Greengrass Core software deletes the oldest Greengrass system log files.

This parameter is equivalent to the log size limit parameter (`totalLogsSizeKB`) of the Greengrass nucleus component. The AWS IoT Greengrass Core software uses the minimum of the two values as the maximum total Greengrass system log size.

`diskSpaceLimitUnit`

(Optional) The unit for the `diskSpaceLimit`. Choose from the following options:

- `KB` – kilobytes
- `MB` – megabytes
- `GB` – gigabytes

Default: `KB`

`deleteLogFileAfterCloudUpload`

(Optional) You can delete a log file after the log manager component uploads the logs to CloudWatch Logs.

Default: `false`

`componentLogsConfigurationMap`

(Optional) A map of log configurations for components on the core device. Each `componentName` object in this map defines the log configuration for the component or application. The Log manager component uploads these component logs to CloudWatch Logs.

> **Important**
>
> We strongly recommend using a single configuration key per component. You should only target a group of files that have only one log file that's actively being written to when using the `logFileRegex`. Not following this recommendation may lead to duplicate logs getting uploaded to CloudWatch. If you are targeting multiple active log files with a single regex, we recommend you upgrade to Log manager v2.3.1 or later and consider changing your configuration using the example configuration.

> **Note**
>
> If you're upgrading from a version of Log manager earlier than v2.2.0, you can continue to use the `componentLogsConfiguration` list instead of `componentLogsConfigurationMap`. However, we strongly recommend that you use the map format so that you can use merge and reset updates to modify configurations for specific components. For information about the `componentLogsConfiguration` parameter, see the configuration parameters for v2.1.x of this component.

`componentName`

The log configuration for the *`componentName`* component or application for this log configuration. You can specify the name of a Greengrass component or another value to identify this log group.

Each object contains the following information:

`minimumLogLevel`

(Optional) The minimum level of log messages to upload. This minimum level applies only if this component's logs use a specific JSON format, which you can find in the AWS IoT Greengrass logging module repository on GitHub.

Choose from the following log levels, listed here in level order:

- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`

Default: `INFO`

`diskSpaceLimit`

(Optional) The maximum total size of all log files for this component, in the unit you specify in `diskSpaceLimitUnit`. After the total size of this component's log files exceeds this maximum total size, the AWS IoT Greengrass Core software deletes this component's oldest log files.

This parameter is related to the log size limit parameter (`totalLogsSizeKB`) of the Greengrass nucleus component. The AWS IoT Greengrass Core software uses the minimum of the two values as the maximum total log size for this component.

> **Note**
>
> The log manager component applies this limit only after it uploads a component's log files to CloudWatch Logs. For a component that writes its own log files, files that aren't uploaded (for example, while the device is offline or uploads fail) aren't deleted, so the total size can exceed this limit.

`diskSpaceLimitUnit`

(Optional) The unit for the `diskSpaceLimit`. Choose from the following options:

- `KB` – kilobytes
- `MB` – megabytes
- `GB` – gigabytes

Default: `KB`

`logFileDirectoryPath`

(Optional) The path to the folder that contains this component's log files.

You don't need to specify this parameter for Greengrass components that print to standard output (stdout) and standard error (stderr).

Default: */greengrass/v2*`/logs`.

`logFileRegex`

(Optional) A regular expression that specifies the log file name format that the component or application uses. The log manager component uses this regular expression to identify log files in the folder at `logFileDirectoryPath`.

You don't need to specify this parameter for Greengrass components that print to standard output (stdout) and standard error (stderr).

If your component or application rotates log files, specify a regex that matches the rotated log file names. For example, you might specify **`hello_world\\\\w*.log`**

to upload logs for a Hello World application. The `\\\\w*` pattern matches zero or more word characters, which includes alphanumeric characters and underscores. This regex matches log files with and without timestamps in their name. In this example, the log manager uploads the following log files:

- `hello_world.log` – The most recent log file for the Hello World application.
- `hello_world_2020_12_15_17_0.log` – An older log file for the Hello World application.

Default: *componentName*`\\\\w*.log`, where *componentName* is the name of the component for this log configuration.

`deleteLogFileAfterCloudUpload`

(Optional) You can delete a log file after the log manager component uploads the logs to CloudWatch Logs.

Default: `false`

`multiLineStartPattern`

(Optional) A regular expression that identifies when a log message on a new line is a new log message. If the regular expression doesn't match the new line, the log manager component appends the new line to the log message for the previous line.

By default, a line begins a new log message only if it starts with a `YYYY-MM-DD` date (optionally after `[` or `(`) or `{`. Otherwise, the log manager component appends that line to the current log message. In v2.3.4 and earlier, any line that doesn't start with a whitespace character is a new log message.

`periodicUploadIntervalSec`

(Optional) The period in seconds at which the log manager component checks for new log files to upload.

Default: `300` (5 minutes)

Minimum: `0.000001` (1 microsecond)

`deprecatedVersionSupport`

Indicates whether the Log manager should use logging speed improvements introduced in Log manager v2.3.5. Set the value to `false` to use the improvements.

If you set this value to `false` when you upgrade from Log manager v2.3.1 or earlier duplicate log entries may be uploaded.

The default is `true`.

Example: Configuration merge update

The following example configuration specifies to upload system logs and `com.example.HelloWorld` component logs to CloudWatch Logs.

```
{
  "logsUploaderConfiguration": {
    "systemLogsConfiguration": {
      "uploadToCloudWatch": "true",
      "minimumLogLevel": "INFO",
      "diskSpaceLimit": "10",
      "diskSpaceLimitUnit": "MB",
      "deleteLogFileAfterCloudUpload": "false"
    },
    "componentLogsConfigurationMap": {
      "com.example.HelloWorld": {
        "minimumLogLevel": "INFO",
        "diskSpaceLimit": "20",
        "diskSpaceLimitUnit": "MB",
        "deleteLogFileAfterCloudUpload": "false"
      }
    }
  },
  "periodicUploadIntervalSec": "300",
  "deprecatedVersionSupport": "false"
}
```

Example: Configuration to upload multiple active log files using Log manager v2.3.1

The following example configuration is the recommended example if you want to target multiple active log files. This example configuration specifies what active log files you want to upload to CloudWatch. Using this configuration example configuration will also upload any rotated files that match the `logFileRegex`. This example configuration is supported on Log manager v2.3.1.

```
{
```

```
  "logsUploaderConfiguration": {
    "componentLogsConfigurationMap": {
      "com.example.A": {
        "logFileRegex": "com.example.A\\w*.log",
        "deleteLogFileAfterCloudUpload": "false"
      }
      "com.example.B": {
        "logFileRegex": "com.example.B\\w*.log",
        "deleteLogFileAfterCloudUpload": "false"
      }
    }
  },
  "periodicUploadIntervalSec": "10"
}
```

v2.3.0 – 2.3.5

`logsUploaderConfiguration`

(Optional) The configuration for logs that the Log manager component uploads. This object contains the following information:

`systemLogsConfiguration`

(Optional) The configuration for AWS IoT Greengrass Core software system logs, which include logs from the Greengrass nucleus and plugin components. Specify this configuration to enable the Log manager component to manage system logs. This object contains the following information:

`uploadToCloudWatch`

(Optional) You can upload system logs to CloudWatch Logs.

Default: `false`

`minimumLogLevel`

(Optional) The minimum level of log messages to upload. This minimum level applies only if you configure the Greengrass nucleus component to output JSON format logs. To enable JSON format logs, specify `JSON` for the logging format parameter (`logging.format`).

Choose from the following log levels, listed here in level order:

- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`

Default: `INFO`

`diskSpaceLimit`

(Optional) The maximum total size of Greengrass system log files, in the unit you specify in `diskSpaceLimitUnit`. After the total size of Greengrass system log files exceeds this maximum total size, the AWS IoT Greengrass Core software deletes the oldest Greengrass system log files.

This parameter is equivalent to the log size limit parameter (`totalLogsSizeKB`) of the Greengrass nucleus component. The AWS IoT Greengrass Core software uses the minimum of the two values as the maximum total Greengrass system log size.

`diskSpaceLimitUnit`

(Optional) The unit for the `diskSpaceLimit`. Choose from the following options:

- `KB` – kilobytes
- `MB` – megabytes
- `GB` – gigabytes

Default: `KB`

`deleteLogFileAfterCloudUpload`

(Optional) You can delete a log file after the log manager component uploads the logs to CloudWatch Logs.

Default: `false`

`componentLogsConfigurationMap`

(Optional) A map of log configurations for components on the core device. Each `componentName` object in this map defines the log configuration for the component or application. The Log manager component uploads these component logs to CloudWatch Logs.

Important

We strongly recommend using a single configuration key per component. You should only target a group of files that have only one log file that's actively being written to when using the `logFileRegex`. Not following this recommendation may lead to duplicate logs getting uploaded to CloudWatch. If you are targeting multiple active log files with a single regex, we recommend you upgrade to Log manager v2.3.1 and consider changing your configuration using the example configuration.

Note

If you're upgrading from a version of Log manager earlier than v2.2.0, you can continue to use the `componentLogsConfiguration` list instead of `componentLogsConfigurationMap`. However, we strongly recommend that you use the map format so that you can use merge and reset updates to modify configurations for specific components. For information about the `componentLogsConfiguration` parameter, see the configuration parameters for v2.1.x of this component.

`componentName`

The log configuration for the *`componentName`* component or application for this log configuration. You can specify the name of a Greengrass component or another value to identify this log group.

Each object contains the following information:

`minimumLogLevel`

(Optional) The minimum level of log messages to upload. This minimum level applies only if this component's logs use a specific JSON format, which you can find in the AWS IoT Greengrass logging module repository on GitHub.

Choose from the following log levels, listed here in level order:

- `DEBUG`
- `INFO`

- `WARN`
- `ERROR`

Default: `INFO`

`diskSpaceLimit`

(Optional) The maximum total size of all log files for this component, in the unit you specify in `diskSpaceLimitUnit`. After the total size of this component's log files exceeds this maximum total size, the AWS IoT Greengrass Core software deletes this component's oldest log files.

This parameter is related to the log size limit parameter (`totalLogsSizeKB`) of the Greengrass nucleus component. The AWS IoT Greengrass Core software uses the minimum of the two values as the maximum total log size for this component.

> **Note**
>
> The log manager component applies this limit only after it uploads a component's log files to CloudWatch Logs. For a component that writes its own log files, files that aren't uploaded (for example, while the device is offline or uploads fail) aren't deleted, so the total size can exceed this limit.

`diskSpaceLimitUnit`

(Optional) The unit for the `diskSpaceLimit`. Choose from the following options:

- `KB` – kilobytes
- `MB` – megabytes
- `GB` – gigabytes

Default: `KB`

`logFileDirectoryPath`

(Optional) The path to the folder that contains this component's log files.

You don't need to specify this parameter for Greengrass components that print to standard output (stdout) and standard error (stderr).

Default: */greengrass/v2*/logs.

`logFileRegex`

(Optional) A regular expression that specifies the log file name format that the component or application uses. The log manager component uses this regular expression to identify log files in the folder at `logFileDirectoryPath`.

You don't need to specify this parameter for Greengrass components that print to standard output (stdout) and standard error (stderr).

If your component or application rotates log files, specify a regex that matches the rotated log file names. For example, you might specify **`hello_world\\\\w*.log`** to upload logs for a Hello World application. The `\\\\w*` pattern matches zero or more word characters, which includes alphanumeric characters and underscores. This regex matches log files with and without timestamps in their name. In this example, the log manager uploads the following log files:

- `hello_world.log` – The most recent log file for the Hello World application.
- `hello_world_2020_12_15_17_0.log` – An older log file for the Hello World application.

Default: *componentName*`\\\\w*.log`, where *componentName* is the name of the component for this log configuration.

`deleteLogFileAfterCloudUpload`

(Optional) You can delete a log file after the log manager component uploads the logs to CloudWatch Logs.

Default: `false`

`multiLineStartPattern`

(Optional) A regular expression that identifies when a log message on a new line is a new log message. If the regular expression doesn't match the new line, the log manager component appends the new line to the log message for the previous line.

By default, a line begins a new log message only if it starts with a `YYYY-MM-DD` date (optionally after `[` or `(`) or `{`. Otherwise, the log manager component

appends that line to the current log message. In v2.3.4 and earlier, any line that doesn't start with a whitespace character is a new log message.

`periodicUploadIntervalSec`

(Optional) The period in seconds at which the log manager component checks for new log files to upload.

Default: `300` (5 minutes)

Minimum: `0.000001` (1 microsecond)

Example: Configuration merge update

The following example configuration specifies to upload system logs and `com.example.HelloWorld` component logs to CloudWatch Logs.

```
{
  "logsUploaderConfiguration": {
    "systemLogsConfiguration": {
      "uploadToCloudWatch": "true",
      "minimumLogLevel": "INFO",
      "diskSpaceLimit": "10",
      "diskSpaceLimitUnit": "MB",
      "deleteLogFileAfterCloudUpload": "false"
    },
    "componentLogsConfigurationMap": {
      "com.example.HelloWorld": {
        "minimumLogLevel": "INFO",
        "diskSpaceLimit": "20",
        "diskSpaceLimitUnit": "MB",
        "deleteLogFileAfterCloudUpload": "false"
      }
    }
  },
  "periodicUploadIntervalSec": "300"
}
```

Example: Configuration to upload multiple active log files using Log manager v2.3.1

The following example configuration is the recommended example if you want to target multiple active log files. This example configuration specifies what active log files you want

to upload to CloudWatch. Using this configuration example configuration will also upload any rotated files that match the `logFileRegex`. This example configuration is supported on Log manager v2.3.1.

```
{
  "logsUploaderConfiguration": {
    "componentLogsConfigurationMap": {
      "com.example.A": {
        "logFileRegex": "com.example.A\\w*.log",
        "deleteLogFileAfterCloudUpload": "false"
      }
      "com.example.B": {
        "logFileRegex": "com.example.B\\w*.log",
        "deleteLogFileAfterCloudUpload": "false"
      }
    }
  },
  "periodicUploadIntervalSec": "10"
}
```

v2.2.x

`logsUploaderConfiguration`

(Optional) The configuration for logs that the Log manager component uploads. This object contains the following information:

`systemLogsConfiguration`

(Optional) The configuration for AWS IoT Greengrass Core software system logs, which include logs from the Greengrass nucleus and plugin components. Specify this configuration to enable the Log manager component to manage system logs. This object contains the following information:

`uploadToCloudWatch`

(Optional) You can upload system logs to CloudWatch Logs.

Default: `false`

`minimumLogLevel`

(Optional) The minimum level of log messages to upload. This minimum level applies only if you configure the Greengrass nucleus component to output JSON format

logs. To enable JSON format logs, specify `JSON` for the logging format parameter (`logging.format`).

Choose from the following log levels, listed here in level order:

- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`

Default: `INFO`

`diskSpaceLimit`

(Optional) The maximum total size of Greengrass system log files, in the unit you specify in `diskSpaceLimitUnit`. After the total size of Greengrass system log files exceeds this maximum total size, the AWS IoT Greengrass Core software deletes the oldest Greengrass system log files.

This parameter is equivalent to the log size limit parameter (`totalLogsSizeKB`) of the Greengrass nucleus component. The AWS IoT Greengrass Core software uses the minimum of the two values as the maximum total Greengrass system log size.

`diskSpaceLimitUnit`

(Optional) The unit for the `diskSpaceLimit`. Choose from the following options:

- `KB` – kilobytes
- `MB` – megabytes
- `GB` – gigabytes

Default: `KB`

`deleteLogFileAfterCloudUpload`

(Optional) You can delete a log file after the log manager component uploads the logs to CloudWatch Logs.

Default: `false`

`componentLogsConfigurationMap`

(Optional) A map of log configurations for components on the core device. Each `componentName` object in this map defines the log configuration for the component or

application. The Log manager component uploads these component logs to CloudWatch Logs.

> **Note**
>
> If you're upgrading from a version of Log manager earlier than v2.2.0, you can continue to use the `componentLogsConfiguration` list instead of `componentLogsConfigurationMap`. However, we strongly recommend that you use the map format so that you can use merge and reset updates to modify configurations for specific components. For information about the `componentLogsConfiguration` parameter, see the configuration parameters for v2.1.x of this component.

`componentName`

The log configuration for the *`componentName`* component or application for this log configuration. You can specify the name of a Greengrass component or another value to identify this log group.

Each object contains the following information:

`minimumLogLevel`

(Optional) The minimum level of log messages to upload. This minimum level applies only if this component's logs use a specific JSON format, which you can find in the AWS IoT Greengrass logging module repository on GitHub.

Choose from the following log levels, listed here in level order:

- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`

Default: `INFO`

`diskSpaceLimit`

(Optional) The maximum total size of all log files for this component, in the unit you specify in `diskSpaceLimitUnit`. After the total size of this component's

log files exceeds this maximum total size, the AWS IoT Greengrass Core software deletes this component's oldest log files.

This parameter is related to the log size limit parameter (`totalLogsSizeKB`) of the Greengrass nucleus component. The AWS IoT Greengrass Core software uses the minimum of the two values as the maximum total log size for this component.

> **Note**
>
> The log manager component applies this limit only after it uploads a component's log files to CloudWatch Logs. For a component that writes its own log files, files that aren't uploaded (for example, while the device is offline or uploads fail) aren't deleted, so the total size can exceed this limit.

`diskSpaceLimitUnit`

(Optional) The unit for the `diskSpaceLimit`. Choose from the following options:

- `KB` – kilobytes
- `MB` – megabytes
- `GB` – gigabytes

Default: `KB`

`logFileDirectoryPath`

(Optional) The path to the folder that contains this component's log files.

You don't need to specify this parameter for Greengrass components that print to standard output (stdout) and standard error (stderr).

Default: *`/greengrass/v2`*`/logs`.

`logFileRegex`

(Optional) A regular expression that specifies the log file name format that the component or application uses. The log manager component uses this regular expression to identify log files in the folder at `logFileDirectoryPath`.

You don't need to specify this parameter for Greengrass components that print to standard output (stdout) and standard error (stderr).

If your component or application rotates log files, specify a regex that matches the rotated log file names. For example, you might specify **`hello_world\\\\w*.log`** to upload logs for a Hello World application. The `\\\\w*` pattern matches zero or more word characters, which includes alphanumeric characters and underscores. This regex matches log files with and without timestamps in their name. In this example, the log manager uploads the following log files:

- `hello_world.log` – The most recent log file for the Hello World application.
- `hello_world_2020_12_15_17_0.log` – An older log file for the Hello World application.

Default: *componentName*`\\\\w*.log`, where *componentName* is the name of the component for this log configuration.

`deleteLogFileAfterCloudUpload`

(Optional) You can delete a log file after the log manager component uploads the logs to CloudWatch Logs.

Default: `false`

`multiLineStartPattern`

(Optional) A regular expression that identifies when a log message on a new line is a new log message. If the regular expression doesn't match the new line, the log manager component appends the new line to the log message for the previous line.

By default, a line begins a new log message only if it starts with a `YYYY-MM-DD` date (optionally after `[` or `(` or `{`. Otherwise, the log manager component appends that line to the current log message. In v2.3.4 and earlier, any line that doesn't start with a whitespace character is a new log message.

`periodicUploadIntervalSec`

(Optional) The period in seconds at which the log manager component checks for new log files to upload.

Default: `300` (5 minutes)

Minimum: `0.000001` (1 microsecond)

Example: Configuration merge update

The following example configuration specifies to upload system logs and `com.example.HelloWorld` component logs to CloudWatch Logs.

```
{
  "logsUploaderConfiguration": {
    "systemLogsConfiguration": {
      "uploadToCloudWatch": "true",
      "minimumLogLevel": "INFO",
      "diskSpaceLimit": "10",
      "diskSpaceLimitUnit": "MB",
      "deleteLogFileAfterCloudUpload": "false"
    },
    "componentLogsConfigurationMap": {
      "com.example.HelloWorld": {
        "minimumLogLevel": "INFO",
        "diskSpaceLimit": "20",
        "diskSpaceLimitUnit": "MB",
        "deleteLogFileAfterCloudUpload": "false"
      }
    }
  },
  "periodicUploadIntervalSec": "300"
}
```

v2.1.x

`logsUploaderConfiguration`

(Optional) The configuration for logs that the Log manager component uploads. This object contains the following information:

`systemLogsConfiguration`

(Optional) The configuration for AWS IoT Greengrass Core software system logs, which include logs from the Greengrass nucleus and plugin components. Specify this configuration to enable the Log manager component to manage system logs. This object contains the following information:

`uploadToCloudWatch`

(Optional) You can upload system logs to CloudWatch Logs.

Default: `false`

`minimumLogLevel`

(Optional) The minimum level of log messages to upload. This minimum level applies only if you configure the Greengrass nucleus component to output JSON format logs. To enable JSON format logs, specify `JSON` for the logging format parameter (`logging.format`).

Choose from the following log levels, listed here in level order:

- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`

Default: `INFO`

`diskSpaceLimit`

(Optional) The maximum total size of Greengrass system log files, in the unit you specify in `diskSpaceLimitUnit`. After the total size of Greengrass system log files exceeds this maximum total size, the AWS IoT Greengrass Core software deletes the oldest Greengrass system log files.

This parameter is equivalent to the log size limit parameter (`totalLogsSizeKB`) of the Greengrass nucleus component. The AWS IoT Greengrass Core software uses the minimum of the two values as the maximum total Greengrass system log size.

`diskSpaceLimitUnit`

(Optional) The unit for the `diskSpaceLimit`. Choose from the following options:

- `KB` – kilobytes
- `MB` – megabytes
- `GB` – gigabytes

Default: `KB`

`deleteLogFileAfterCloudUpload`

(Optional) You can delete a log file after the log manager component uploads the logs to CloudWatch Logs.

Default: `false`

`componentLogsConfiguration`

(Optional) A list of log configurations for components on the core device. Each configuration in this list defines the log configuration for a component or application. The Log manager component uploads these component logs to CloudWatch Logs

Each object contains the following information:

`componentName`

The name of the component or application for this log configuration. You can specify the name of a Greengrass component or another value to identify this log group.

`minimumLogLevel`

(Optional) The minimum level of log messages to upload. This minimum level applies only if this component's logs use a specific JSON format, which you can find in the AWS IoT Greengrass logging module repository on GitHub.

Choose from the following log levels, listed here in level order:

- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`

Default: `INFO`

`diskSpaceLimit`

(Optional) The maximum total size of all log files for this component, in the unit you specify in `diskSpaceLimitUnit`. After the total size of this component's log files exceeds this maximum total size, the AWS IoT Greengrass Core software deletes this component's oldest log files.

This parameter is related to the log size limit parameter (`totalLogsSizeKB`) of the Greengrass nucleus component. The AWS IoT Greengrass Core software uses the minimum of the two values as the maximum total log size for this component.

> **Note**
>
> The log manager component applies this limit only after it uploads a component's log files to CloudWatch Logs. For a component that writes its own log files, files that aren't uploaded (for example, while the device is offline or uploads fail) aren't deleted, so the total size can exceed this limit.

`diskSpaceLimitUnit`

(Optional) The unit for the `diskSpaceLimit`. Choose from the following options:

- `KB` – kilobytes
- `MB` – megabytes
- `GB` – gigabytes

Default: `KB`

`logFileDirectoryPath`

(Optional) The path to the folder that contains this component's log files.

You don't need to specify this parameter for Greengrass components that print to standard output (stdout) and standard error (stderr).

Default: */greengrass/v2*`/logs`.

`logFileRegex`

(Optional) A regular expression that specifies the log file name format that the component or application uses. The log manager component uses this regular expression to identify log files in the folder at `logFileDirectoryPath`.

You don't need to specify this parameter for Greengrass components that print to standard output (stdout) and standard error (stderr).

If your component or application rotates log files, specify a regex that matches the rotated log file names. For example, you might specify **`hello_world\\\\w*.log`** to upload logs for a Hello World application. The `\\\\w*` pattern matches zero or more word characters, which includes alphanumeric characters and underscores. This regex matches log files with and without timestamps in their name. In this example, the log manager uploads the following log files:

- `hello_world.log` – The most recent log file for the Hello World application.
- `hello_world_2020_12_15_17_0.log` – An older log file for the Hello World application.

Default: *`componentName`*`\\\\w*.log`, where *`componentName`* is the name of the component for this log configuration.

`deleteLogFileAfterCloudUpload`

(Optional) You can delete a log file after the log manager component uploads the logs to CloudWatch Logs.

Default: `false`

`multiLineStartPattern`

(Optional) A regular expression that identifies when a log message on a new line is a new log message. If the regular expression doesn't match the new line, the log manager component appends the new line to the log message for the previous line.

By default, a line begins a new log message only if it starts with a `YYYY-MM-DD` date (optionally after `[` or `(` or `{`. Otherwise, the log manager component appends that line to the current log message. In v2.3.4 and earlier, any line that doesn't start with a whitespace character is a new log message.

`periodicUploadIntervalSec`

(Optional) The period in seconds at which the log manager component checks for new log files to upload.

Default: `300` (5 minutes)

Minimum: `0.000001` (1 microsecond)

Example: Configuration merge update

The following example configuration specifies to upload system logs and `com.example.HelloWorld` component logs to CloudWatch Logs.

```
{
  "logsUploaderConfiguration": {
    "systemLogsConfiguration": {
```

```
      "uploadToCloudWatch": "true",
      "minimumLogLevel": "INFO",
      "diskSpaceLimit": "10",
      "diskSpaceLimitUnit": "MB",
      "deleteLogFileAfterCloudUpload": "false"
    },
    "componentLogsConfiguration": [
      {
        "componentName": "com.example.HelloWorld",
        "minimumLogLevel": "INFO",
        "diskSpaceLimit": "20",
        "diskSpaceLimitUnit": "MB",
        "deleteLogFileAfterCloudUpload": "false"
      }
    ]
  },
  "periodicUploadIntervalSec": "300"
}
```

v2.0.x

`logsUploaderConfiguration`

(Optional) The configuration for logs that the Log manager component uploads. This object contains the following information:

`systemLogsConfiguration`

(Optional) The configuration for AWS IoT Greengrass Core software system logs. Specify this configuration to enable the Log manager component to manage system logs. This object contains the following information:

`uploadToCloudWatch`

(Optional) You can upload system logs to CloudWatch Logs.

Default: `false`

`minimumLogLevel`

(Optional) The minimum level of log messages to upload. This minimum level applies only if you configure the Greengrass nucleus component to output JSON format logs. To enable JSON format logs, specify `JSON` for the logging format parameter (`logging.format`).

Choose from the following log levels, listed here in level order:

- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`

Default: `INFO`

`diskSpaceLimit`

(Optional) The maximum total size of Greengrass system log files, in the unit you specify in `diskSpaceLimitUnit`. After the total size of Greengrass system log files exceeds this maximum total size, the AWS IoT Greengrass Core software deletes the oldest Greengrass system log files.

This parameter is equivalent to the log size limit parameter (`totalLogsSizeKB`) of the Greengrass nucleus component. The AWS IoT Greengrass Core software uses the minimum of the two values as the maximum total Greengrass system log size.

`diskSpaceLimitUnit`

(Optional) The unit for the `diskSpaceLimit`. Choose from the following options:

- `KB` – kilobytes
- `MB` – megabytes
- `GB` – gigabytes

Default: `KB`

`deleteLogFileAfterCloudUpload`

(Optional) You can delete a log file after the log manager component uploads the logs to CloudWatch Logs.

Default: `false`

`componentLogsConfiguration`

(Optional) A list of log configurations for components on the core device. Each configuration in this list defines the log configuration for a component or application. The Log manager component uploads these component logs to CloudWatch Logs

Each object contains the following information:

`componentName`

The name of the component or application for this log configuration. You can specify the name of a Greengrass component or another value to identify this log group.

`minimumLogLevel`

(Optional) The minimum level of log messages to upload. This minimum level applies only if this component's logs use a specific JSON format, which you can find in the AWS IoT Greengrass logging module repository on GitHub.

Choose from the following log levels, listed here in level order:

- `DEBUG`
- `INFO`
- `WARN`
- `ERROR`

Default: `INFO`

`diskSpaceLimit`

(Optional) The maximum total size of all log files for this component, in the unit you specify in `diskSpaceLimitUnit`. After the total size of this component's log files exceeds this maximum total size, the AWS IoT Greengrass Core software deletes this component's oldest log files.

This parameter is related to the log size limit parameter (`totalLogsSizeKB`) of the Greengrass nucleus component. The AWS IoT Greengrass Core software uses the minimum of the two values as the maximum total log size for this component.

> **Note**
>
> The log manager component applies this limit only after it uploads a component's log files to CloudWatch Logs. For a component that writes its own log files, files that aren't uploaded (for example, while the device is offline or uploads fail) aren't deleted, so the total size can exceed this limit.

`diskSpaceLimitUnit`

(Optional) The unit for the `diskSpaceLimit`. Choose from the following options:

- KB – kilobytes
- MB – megabytes
- GB – gigabytes

Default: KB

`logFileDirectoryPath`

The path to the folder that contains this component's log files.

To upload a Greengrass component's logs, specify ***/greengrass/v2*****/logs**, and replace */greengrass/v2* with your Greengrass root folder.

`logFileRegex`

A regular expression that specifies the log file name format that the component or application uses. The Log manager component uses this regular expression to identify log files in the folder at `logFileDirectoryPath`.

To upload a Greengrass component's logs, specify a regex that matches the rotated log file names. For example, you might specify **`com.example.HelloWorld\\w*.log`** to upload logs for a Hello World component. The `\\w*` pattern matches zero or more word characters, which includes alphanumeric characters and underscores. This regex matches log files with and without timestamps in their name. In this example, the Log manager uploads the following log files:

- `com.example.HelloWorld.log` – The most recent log file for the Hello World component.
- `com.example.HelloWorld_2020_12_15_17_0.log` – An older log file for the Hello World component. The Greengrass nucleus adds a rotating timestamp to the log files.

`deleteLogFileAfterCloudUpload`

(Optional) You can delete a log file after the log manager component uploads the logs to CloudWatch Logs.

Default: `false`

`multiLineStartPattern`

(Optional) A regular expression that identifies when a log message on a new line is a new log message. If the regular expression doesn't match the new line, the log manager component appends the new line to the log message for the previous line.

By default, a line begins a new log message only if it starts with a `YYYY-MM-DD` date (optionally after `[` or `(` or `{`. Otherwise, the log manager component appends that line to the current log message. In v2.3.4 and earlier, any line that doesn't start with a whitespace character is a new log message.

`periodicUploadIntervalSec`

(Optional) The period in seconds at which the log manager component checks for new log files to upload.

Default: `300` (5 minutes)

Minimum: `0.000001` (1 microsecond)

Example: Configuration merge update

The following example configuration specifies to upload system logs and `com.example.HelloWorld` component logs to CloudWatch Logs.

```
{
  "logsUploaderConfiguration": {
    "systemLogsConfiguration": {
      "uploadToCloudWatch": "true",
      "minimumLogLevel": "INFO",
      "diskSpaceLimit": "10",
      "diskSpaceLimitUnit": "MB",
      "deleteLogFileAfterCloudUpload": "false"
    },
    "componentLogsConfiguration": [
      {
        "componentName": "com.example.HelloWorld",
        "minimumLogLevel": "INFO",
        "logFileDirectoryPath": "/greengrass/v2/logs",
        "logFileRegex": "com.example.HelloWorld\\w*.log",
        "diskSpaceLimit": "20",
        "diskSpaceLimitUnit": "MB",
```

```
        "deleteLogFileAfterCloudUpload": "false"
      }
    ]
  },
  "periodicUploadIntervalSec": "300"
}
```

Usage

The Log manager component uploads to the following log groups and log streams.

2.1.0 and later

Log group name

```
/aws/greengrass/componentType/region/componentName
```

The log group name uses the following variables:

- `componentType` – The type of the component, which can be one of the following:
  - `GreengrassSystemComponent` – This log group includes logs for the nucleus and plugin components, which run in the same JVM as the Greengrass nucleus. The component is part of the Greengrass nucleus.
  - `UserComponent` – This log group includes logs for generic components, Lambda components, and other applications on the device. The component isn't part of the Greengrass nucleus.

  For more information, see Component types.
- `region` – The AWS Region that the core device uses.
- `componentName` – The name of the component. For system logs, this value is `System`.

Log stream name

```
/date/thing/thingName
```

The log stream name uses the following variables:

- `date` – The date of the log, such as `2020/12/15`. The log manager component uses the `yyyy/MM/dd` format.

- `thingName` – The name of the core device.

> **Note**
>
> If a thing name contains a colon (`:`), the log manager replaces the colon with a plus (`+`).

2.0.x

Log group name

```
/aws/greengrass/componentType/region/componentName
```

The log group name uses the following variables:

- `componentType` – The type of the component, which can be one of the following:
  - `GreengrassSystemComponent` – The component is part of the Greengrass nucleus.
  - `UserComponent` – The component isn't part of the Greengrass nucleus. The Log manager uses this type for Greengrass components and other applications on the device.
- `region` – The AWS Region that the core device uses.
- `componentName` – The name of the component. For system logs, this value is `System`.

Log stream name

```
/date/deploymentTargets/thingName
```

The log stream name uses the following variables:

- `date` – The date of the log, such as `2020/12/15`. The Log manager component uses the `yyyy/MM/dd` format.
- `deploymentTargets` – The things whose deployments include the component. The Log manager component separates each target by a slash. If the component runs on the core device as the result of a local deployment, this value is `LOCAL_DEPLOYMENT`.

  Consider an example where you have a core device named `MyGreengrassCore`, and the core device has two deployments:

  - A deployment that targets the core device, `MyGreengrassCore`.

- A deployment that targets a thing group named `MyGreengrassCoreGroup`, which contains the core device.

  The `deploymentTargets` for this core device are `thing/MyGreengrassCore/thinggroup/MyGreengrassCoreGroup`.
- `thingName` – The name of the core device.

Formats for log entries.

The Greengrass nucleus writes log files in either string or JSON format. For system logs, you control the format by setting the `format` field of the `logging` entry. You can find the `logging` entry in the Greengrass nucleus component's configuration file. For more information, see Greengrass nucleus configuration.

The text format is free-form and accepts any string. The following fleet status service message is an example of string formatted logging:

```
2023-03-26T18:18:27.271Z [INFO] (pool-1-thread-2)
com.aws.greengrass.status.FleetStatusService: fss-status-update-published.
Status update published to FSS. {trigger=CADENCE, serviceName=FleetStatusService,
currentState=RUNNING}
```

You should use the JSON format if you want to view logs with the Greengrass CLI logs command or interact with logs programmatically. The following example outlines the JSON shape:

```
{
   "loggerName": <string>,
   "level": <"DEBUG" | "INFO" | "ERROR" | "TRACE" | "WARN">,
   "eventType": <string, optional>,
   "cause": <string, optional>,
   "contexts": {},
   "thread": <string>,
   "message": <string>,
   "timestamp": <epoch time> # Needs to be epoch time
}
```

To control the output of your component's logs, you can use the `minimumLogLevel` configuration option. To use this option, your component must write its log entries in JSON format. You should use the same format as the system log file.

Local log file

This component uses the same log file as the Greengrass nucleus component.

Linux

```
/greengrass/v2/logs/greengrass.log
```

Windows

```
C:\greengrass\v2\logs\greengrass.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
  ```

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.3.14	Bug fixes and improvements • Fixes an issue where logs with multibyte UTF-8 characters (such as Japanese, Chinese, or emoji) cause duplicate uploads to Amazon CloudWatch.

Version	Changes
2.3.13	Updates the component version for the Greengrass nucleus version 2.18.0 release.
2.3.12	Updates the component version for the Greengrass nucleus version 2.17.0 release.
2.3.11	Bug fixes and improvements • Fixes an issue where Log manager runtime configuration grew indefinit ely with stale information of uploaded log files.
2.3.10	New features • Adds a new configuration key (`updateToTlogIntervalSec`) to control the frequency at which log-upload event details are persisted to the local transaction log (`config.tlog`). Bug fixes and improvements • Improves Log manager to refresh cloudwatch client for socket connection error.
2.3.9	Version updated for Greengrass nucleus version 2.14.0 release.
2.3.8	Version updated for Greengrass nucleus version 2.13.0 release.
2.3.7	Version updated for Greengrass nucleus version 2.12.0 release.
2.3.6	Bug fixes and improvements • Adjusts log levels for certain errors.
2.3.5	Improvements Improves log upload speed. Version updated for Greengrass nucleus version 2.11.0 release.

Version	Changes
2.3.4	Bug fixes and improvements • Adds support for setting the `periodicUploadIntervalSec` parameter to fractional values. The minimum is 1 microsecond. • Fixes an issue where log manager doesn't respect the CloudWatch `putLogEvents` limits.
2.3.3	Version updated for Greengrass nucleus version 2.10.0 release.
2.3.2	Bug fixes and improvements • Improves space management so that log files are not deleted before they are uploaded. • Fixes issues with cache management. • Additional minor bug fixes and improvements.
2.3.1	Bug fixes and improvements • Fixes an issue where s that target file groups with multiples active log files upload duplicate entries to CloudWatch. • Additional minor bug fixes and improvements.
2.3.0	**Note** We recommend that you upgrade to Greengrass nucleus 2.9.1 when you upgrade to Log manager 2.3.0. New features Reduces log delays by processing and directly uploading active log files instead of waiting for new files to be rotated. Bug fixes and improvements • Improves support of log rotation when rotating files with a unique name. • Additional minor bug fixes and improvements.

Version	Changes
2.2.8	Version updated for Greengrass nucleus version 2.9.0 release.
2.2.7	Version updated for Greengrass nucleus version 2.8.0 release.
2.2.6	Version updated for Greengrass nucleus version 2.7.0 release.
2.2.5	Version updated for Greengrass nucleus version 2.6.0 release.
2.2.4	Bug fixes and improvements • Improves stability when handling invalid configurations. • Additional minor fixes and improvements.
2.2.3	Bug fixes and improvements • Improves stability in certain scenarios where the component restarts or encounters errors. • Fixes issues where large log messages and large log files fail to upload in certain scenarios. • Fixes issues with how this component handles configuration reset updates. • Fixes an issue where a `null` `diskSpaceLimit` configuration value prevented the component from deploying.
2.2.2	Bug fixes and improvements • Adds support for log messages that are larger than 256 kilobytes. The Log manager component splits these large log messages into multiple messages with the same log event timestamp.
2.2.1	Version updated for Greengrass nucleus version 2.5.0 release.
2.2.0	New feature • Adds the `componentLogsConfigurationMap` configuration parameter to support a map format for component log configurations. Each `componentName` object in the map defines the log configura tion for a component or application.

Version	Changes
2.1.3	Version updated for Greengrass nucleus version 2.4.0 release.
2.1.2	Version updated for Greengrass nucleus version 2.3.0 release.
2.1.1	Bug fixes and improvements • Fixes an issue where the system log configuration wasn't updated in certain cases.
2.1.0	Bug fixes and improvements • Use defaults for `logFileDirectoryPath` and `logFileRegex` that work for Greengrass components that print to standard output (stdout) and standard error (stderr). • Correctly route traffic through a configured network proxy when uploading logs to CloudWatch Logs. • Correctly handle colon characters (`:`) in log stream names. CloudWatch Logs log stream names don't support colons. • Simplify log stream names by removing thing group names from the log stream. • Remove an error log message that prints during normal behavior.
2.0.x	Initial version.

Machine learning components

AWS IoT Greengrass provides the following machine learning components that you can deploy to supported devices to perform machine learning inference using models trained in Amazon SageMaker AI or with your own pre-trained models that are stored in Amazon S3.

AWS provides the following categories of machine learning components:

- **Model component**—Contains machine learning models as Greengrass artifacts.
- **Runtime component**—Contains the script that installs the machine learning framework and its dependencies on the Greengrass core device.

- **Inference component**—Contains the inference code and includes component dependencies to install the machine learning framework and download pre-trained machine learning models.

You can use the sample inference code and pre-trained models in the AWS-provided machine learning components to perform image classification and object detection using DLR and TensorFlow Lite. To perform custom machine learning inference with your own models that are stored in Amazon S3, or to use a different machine learning framework, you can use the recipes of these public components as templates to create custom machine learning components. For more information, see Customize your machine learning components.

AWS IoT Greengrass also includes an AWS-provided component to manage the installation and lifecycle of the SageMaker AI Edge Manager agent on Greengrass core devices. With SageMaker AI Edge Manager, you can use Amazon SageMaker AI Neo-compiled models directly on your core device. For more information, see Use Amazon SageMaker AI Edge Manager on Greengrass core devices.

The following table lists the machine learning components that are available in AWS IoT Greengrass.

> **Note**
>
> Several AWS-provided components depend on specific minor versions of the Greengrass nucleus. Because of this dependency, you need to update these components when you update the Greengrass nucleus to a new minor version. For information about the specific versions of the nucleus that each component depends on, see the corresponding component topic. For more information about updating the nucleus, see Update the AWS IoT Greengrass Core software (OTA).

When a component has a component type of both generic and Lambda, the current version of the component is the generic type and a previous version of the component is the Lambda type.

Component	Description	Component type	Supported OS	Open source
SageMaker AI Edge Manager	Deploys the Amazon SageMaker AI	Generic	Linux, Windows	No

Component	Description	Component type	Supported OS	Open source
	Edge Manager agent on the Greengrass core device.			
DLR image classification	Inference component that uses the DLR image classification model store and the DLR runtime component as dependencies to install DLR, download sample image classification models, and perform image classification inference on supported devices.	Generic	Linux, Windows	No

Component	Description	Component type	Supported OS	Open source
DLR object detection	Inference component that uses the DLR object detection model store and the DLR runtime component as dependencies to install DLR, download sample object detection models, and perform object detection inference on supported devices.	Generic	Linux, Windows	No
DLR image classification model store	Model component that contains sample ResNet-50 image classification models as Greengrass artifacts.	Generic	Linux, Windows	No

Component	Description	Component type	Supported OS	Open source
DLR object detection model store	Model component that contains sample YOLOv3 object detection models as Greengrass artifacts.	Generic	Linux, Windows	No
DLR runtime	Runtime component that contains an installation script that is used to install DLR and its dependenc ies on the Greengrass core device.	Generic	Linux, Windows	No

Component	Description	Component type	Supported OS	Open source
TensorFlow Lite image classification	Inference component that uses the TensorFlow Lite image classific ation model store and the TensorFlow Lite runtime component as dependenc ies to install TensorFlow Lite, download sample image classification models, and perform image classification inference on supported devices.	Generic	Linux, Windows	No

Component	Description	Component type	Supported OS	Open source
TensorFlow Lite object detection	Inference component that uses the TensorFlow Lite object detection model store and the TensorFlow Lite runtime component as dependencies to install TensorFlow Lite, download sample object detection models, and perform object detection inference on supported devices.	Generic	Linux, Windows	No
TensorFlow Lite image classification model store	Model component that contains a sample MobileNet v1 model as a Greengrass artifact.	Generic	Linux, Windows	No

Component	Description	Component type	Supported OS	Open source
TensorFlow Lite object detection model store	Model component that contains a sample Single Shot Detection (SSD) MobileNet model as a Greengrass artifact.	Generic	Linux, Windows	No
TensorFlow Lite runtime	Runtime component that contains an installation script that is used to install TensorFlow Lite and its dependencies on the Greengrass core device.	Generic	Linux, Windows	No

SageMaker AI Edge Manager

Important

SageMaker AI Edge Manager was discontinued on April 26th, 2024. For more information about continuing to deploy your models to edge devices, see SageMaker AI Edge Manager end of life.

The Amazon SageMaker AI Edge Manager component (`aws.greengrass.SageMakerEdgeManager`) installs the SageMaker AI Edge Manager agent binary.

SageMaker AI Edge Manager provides model management for edge devices so you can optimize, secure, monitor, and maintain machine learning models on fleets of edge devices. The SageMaker AI Edge Manager component installs and manages the lifecycle of the SageMaker AI Edge Manager agent on your core device. You can also use SageMaker AI Edge Manager to package and use SageMaker AI Neo-compiled models as model components on Greengrass core devices. For more information about using SageMaker AI Edge Manager agent on your core device, see Use Amazon SageMaker AI Edge Manager on Greengrass core devices.

SageMaker AI Edge Manager component v1.3.x installs Edge Manager agent binary v1.20220822.836f3023. For more information about Edge Manager agent binary versions, see Edge Manager Agent.

Note

The SageMaker AI Edge Manager component is available only in the following AWS Regions:

- US East (Ohio)
- US East (N. Virginia)
- US West (Oregon)
- EU (Frankfurt)
- EU (Ireland)
- Asia Pacific (Tokyo)

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration

- Local log file
- Changelog

Versions

This component has the following versions:

- 1.3.x
- 1.2.x
- 1.1.x
- 1.0.x

Type

This component is a generic component (`aws.greengrass.generic`). The Greengrass nucleus runs the component's lifecycle scripts.

For more information, see Component types.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

Requirements

This component has the following requirements:

- A Greengrass core device running on Amazon Linux 2, a Debian-based Linux platform (x86_64 or Armv8), or Windows (x86_64). If you don't have one, see Tutorial: Getting started with AWS IoT Greengrass V2.
- Python 3.6 or later, including `pip` for your version of Python, installed on your core device.
- The Greengrass device role configured with the following:
  - A trust relationship that allows `credentials.iot.amazonaws.com` and `sagemaker.amazonaws.com` to assume the role, as shown in the following IAM policy example.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "credentials.iot.amazonaws.com"
       },
      "Action": "sts:AssumeRole"
    },
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "sagemaker.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

- The AmazonSageMakerEdgeDeviceFleetPolicy IAM managed policy.
- The `s3:PutObject` action, as shown in the following IAM policy example.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "s3:PutObject"
      ],
      "Resource": [
        "*"
      ],
      "Effect": "Allow"
    }
  ]
}
```

- An Amazon S3 bucket created in the same AWS account and AWS Region as your Greengrass core device. SageMaker AI Edge Manager requires an S3 bucket to create an edge device fleet, and

to store sample data from running inference on your device. For information about creating S3 buckets, see Getting started with Amazon S3.

- A SageMaker AI edge device fleet that uses the same AWS IoT role alias as your Greengrass core device. For more information, see Create an edge device fleet.
- Your Greengrass core device registered as an edge device in your SageMaker AI Edge device fleet. The edge device name must match the AWS IoT thing name for your core device. For more information, see Register your Greengrass core device.

Endpoints and ports

This component must be able to perform outbound requests to the following endpoints and ports, in addition to endpoints and ports required for basic operation. For more information, see Allow device traffic through a proxy or firewall.

Endpoint	Port	Required	Description
`edge.sagemaker.`*`region`*`.amazonaws.com`	443	Yes	Check device registration status and send metrics to SageMaker AI.
`*.s3.amazonaws.com`	443	Yes	Upload capture data to the S3 bucket that you specify. You can replace * with the name

Endpoint	Port	Required	Description
			of each bucket where you upload data.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

1.3.5 and 1.3.6

The following table lists the dependencies for version 1.3.5 and 1.3.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.13.0	Soft
Token exchange service	>=0.0.0	Hard

1.3.4

The following table lists the dependencies for version 1.3.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.12.0	Soft
Token exchange service	>=0.0.0	Hard

1.3.3

The following table lists the dependencies for version 1.3.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.11.0	Soft
Token exchange service	>=0.0.0	Hard

1.3.2

The following table lists the dependencies for version 1.3.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.10.0	Soft
Token exchange service	>=0.0.0	Hard

1.3.1

The following table lists the dependencies for version 1.3.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.9.0	Soft
Token exchange service	>=0.0.0	Hard

1.1.1 - 1.3.0

The following table lists the dependencies for versions 1.1.1 - 1.3.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.8.0	Soft

Dependency	Compatible versions	Dependency type
Token exchange service	>=0.0.0	Hard

1.1.0

The following table lists the dependencies for version 1.1.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.6.0	Soft
Token exchange service	>=0.0.0	Hard

1.0.3

The following table lists the dependencies for version 1.0.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.5.0	Soft
Token exchange service	>=0.0.0	Hard

1.0.1 and 1.0.2

The following table lists the dependencies for versions 1.0.1 and 1.0.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.4.0	Soft
Token exchange service	>=0.0.0	Hard

1.0.0

The following table lists the dependencies for version 1.0.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.3.0	Soft
Token exchange service	>=0.0.0	Hard

For more information about component dependencies, see the component recipe reference.

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

> **Note**
>
> This section describes the configuration parameters that you set in the component. For more information about the corresponding SageMaker AI Edge Manager configuration, see Edge Manager Agent in the *Amazon SageMaker AI Developer Guide*.

`DeviceFleetName`

The name of the SageMaker AI Edge Manager device fleet that contains your Greengrass core device.

You must specify a value for this parameter in the configuration update when you deploy this component.

`BucketName`

The name of the S3 bucket to which you upload captured inference data. The bucket name must contain the string `sagemaker`.

If you set `CaptureDataDestination` to `Cloud`, or if you set `CaptureDataPeriodicUpload` to `true`, then you must specify a value for this parameter in the configuration update when you deploy this component.

> **Note**
>
> Capture data is an SageMaker AI feature that you use to upload inference input, inference results, and additional inference data to an S3 bucket or a local directory for future analysis. For more information about using capture data with SageMaker AI Edge Manager, see Manage Model in the *Amazon SageMaker AI Developer Guide*.

`CaptureDataBatchSize`

(Optional) The size of a batch of capture data requests that the agent handles. This value must be less than the buffer size that you specify in `CaptureDataBufferSize`. We recommend that you don't exceed half the buffer size.

The agent handles a request batch when the number of requests in the buffer meets the `CaptureDataBatchSize` number, or when the `CaptureDataPushPeriodSeconds` interval elapses, whichever occurs first.

Default: `10`

`CaptureDataBufferSize`

(Optional) The maximum number of capture data requests stored in the buffer.

Default: `30`

`CaptureDataDestination`

(Optional) The destination where you store captured data. This parameter can have the following values:

- `Cloud`—Uploads captured data to the S3 bucket that you specify in `BucketName`.
- `Disk`—Writes captured data to the component's work directory.

If you specify `Disk`, you can also choose to periodically upload the captured data to your S3 bucket by setting `CaptureDataPeriodicUpload` to `true`.

Default: `Cloud`

`CaptureDataPeriodicUpload`

(Optional) String value that specifies whether to periodically upload captured data. Supported values are `true` and `false`.

Set this parameter to `true` if you set `CaptureDataDestination` to `Disk`, and you also want the agent to periodically upload the captured data your S3 bucket.

Default: `false`

`CaptureDataPeriodicUploadPeriodSeconds`

(Optional) The interval in seconds at which SageMaker AI Edge Manager agent uploads captured data to the S3 bucket. Use this parameter if you set `CaptureDataPeriodicUpload` to `true`.

Default: 8

`CaptureDataPushPeriodSeconds`

(Optional) The interval in seconds at which SageMaker AI Edge Manager agent handles a batch of capture data requests from the buffer.

The agent handles a request batch when the number of requests in the buffer meets the `CaptureDataBatchSize` number, or when the `CaptureDataPushPeriodSeconds` interval elapses, whichever occurs first.

Default: 4

`CaptureDataBase64EmbedLimit`

(Optional) The maximum size in bytes of captured data that SageMaker AI Edge Manager agent uploads.

Default: `3072`

`FolderPrefix`

(Optional) The name of the folder to which the agent writes the captured data. If you set `CaptureDataDestination` to `Disk`, the agent creates the folder in the directory that is specified by `CaptureDataDiskPath`. If you set `CaptureDataDestination` to `Cloud`, or if you set `CaptureDataPeriodicUpload` to `true`, the agent creates the folder in your S3 bucket.

Default: `sme-capture`

`CaptureDataDiskPath`

This feature is available in v1.1.0 and later versions of the SageMaker AI Edge Manager component.

(Optional) The path to the folder to which the agent creates the captured data folder. If you set `CaptureDataDestination` to `Disk`, the agent creates the captured data folder in this directory. If you don't specify this value, the agent creates the captured data folder in the component's work directory. Use the `FolderPrefix` parameter to specify the name of the captured data folder.

Default: */greengrass/v2*`/work/aws.greengrass.SageMakerEdgeManager/capture`

`LocalDataRootPath`

This feature is available in v1.2.0 and later versions of the SageMaker AI Edge Manager component.

(Optional) The path where this component stores the following data on the core device:

- The local database for runtime data when you set `DbEnable` to `true`.
- SageMaker AI Neo-compiled models that this component automatically downloads when you set `DeploymentEnable` to `true`.

Default: */greengrass/v2*`/work/aws.greengrass.SageMakerEdgeManager`

`DbEnable`

(Optional) You can enable this component to store runtime data in a local database to preserve the data, in case the component fails or the device loses power.

This database requires 5 MB of storage on the core device's file system.

Default: `false`

`DeploymentEnable`

This feature is available in v1.2.0 and later versions of the SageMaker AI Edge Manager component.

(Optional) You can enable this component to automatically retrieve SageMaker AI Neo-compiled models from that you upload to Amazon S3. After you upload a new model to Amazon S3, use SageMaker AI Studio or the SageMaker AI API to deploy the new model to this core device. When you enable this feature, you can deploy new models to core devices without needing to create a AWS IoT Greengrass deployment.

> **Important**
>
> To use this feature, you must set `DbEnable` to `true`. This feature uses the local database to track models that it retrieves from the AWS Cloud.

Default: `false`

`DeploymentPollInterval`

This feature is available in v1.2.0 and later versions of the SageMaker AI Edge Manager component.

(Optional) The amount of time (in minutes) between which this component checks for new models to download. This option applies when you set `DeploymentEnable` to `true`.

Default: `1440` (1 day)

`DLRBackendOptions`

This feature is available in v1.2.0 and later versions of the SageMaker AI Edge Manager component.

(Optional) The DLR runtime flags to set in the DLR runtime that this component uses. You can set the following flag:

- `TVM_TENSORRT_CACHE_DIR` – Enable TensorRT model caching. Specify an absolute path to an existing folder that has read/write permissions.
- `TVM_TENSORRT_CACHE_DISK_SIZE_MB` – Assigns the upper limit of the TensorRT model cache folder. When the directory size grows beyond this limit the cached engines that are used the least are deleted. The default value is 512 MB.

For example, you can set this parameter to the following value to enable TensorRT model caching and limit the cache size to 800 MB.

```
TVM_TENSORRT_CACHE_DIR=/data/secured_folder/trt/cache;
 TVM_TENSORRT_CACHE_DISK_SIZE_MB=800
```

`SagemakerEdgeLogVerbose`

(Optional) String value that specifies whether to enable debug logging. Supported values are `true` and `false`.

Default: `false`

`UnixSocketName`

(Optional) The location of the SageMaker AI Edge Manager socket file descriptor on the core device.

Default: `/tmp/aws.greengrass.SageMakerEdgeManager.sock`

Example: Configuration merge update

The following example configuration specifies that the core device is part of the *MyEdgeDeviceFleet* and that the agent writes capture data both to the device and to an S3 bucket. This configuration also enables debug logging.

```
{
    "DeviceFleetName": "MyEdgeDeviceFleet",
    "BucketName": "amzn-s3-demo-bucket",
    "CaptureDataDestination": "Disk",
    "CaptureDataPeriodicUpload": "true",
    "SagemakerEdgeLogVerbose": "true"
}
```

Local log file

This component uses the following log file.

Linux

```
/greengrass/v2/logs/aws.greengrass.SageMakerEdgeManager.log
```

Windows

```
C:\greengrass\v2\logs\aws.greengrass.SageMakerEdgeManager.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

Linux

```
sudo tail -f /greengrass/v2/logs/aws.greengrass.SageMakerEdgeManager.log
```

Windows (PowerShell)

```
Get-Content C:\greengrass\v2\logs\aws.greengrass.SageMakerEdgeManager.log -Tail
 10 -Wait
```

Changelog

The following table describes the changes in each version of the component.

Version	Changes
1.3.6	Version updated for Greengrass nucleus 2.12.5 release.
1.3.5	Version updated for Greengrass nucleus version 2.12.0 release.
1.3.4	Version updated for Greengrass nucleus version 2.11.0 release.
1.3.3	Version updated for Greengrass nucleus version 2.10.0 release.
1.3.2	Version updated for Greengrass nucleus version 2.9.0 release.
1.3.1	Version updated for Greengrass nucleus version 2.8.0 release.
1.3.0	New features • Adds support for TensorRT cache disk size management. • Adds the optional `TVM_TENSORRT_CACHE_DISK_SIZE_MB` flag to the DLRBackendOptions parameter to set the size limit for cached models on disk. Improvements • Provides improved prediction concurrency. This helps to get better usage of device accelerator engines, such as GPUs.

Version	Changes
1.2.0	New features • Adds support for this component to automatically retrieve SageMaker AI Neo-compiled models that you upload to Amazon S3. When you enable this feature, you can deploy new models to core devices without needing to create a AWS IoT Greengrass deployment. • Adds support for a backup database that this component uses to preserve runtime data, in case the component fails or the device loses power. • Adds support for you to configure DLR runtime flags when you configure this component.
1.1.1	Version updated for Greengrass nucleus version 2.7.0 release.
1.1.0	New features • Adds support for Greengrass core devices running Amazon Linux 2. • Adds the new `CaptureDataDiskPath` configuration parameter. You can use this parameter to specify the path of the captured data folder on your device. Bug fixes and improvements • Version updated for Greengrass nucleus version 2.5.0 release.
1.0.3	Version updated for Greengrass nucleus version 2.4.0 release.
1.0.2	Bug fixes and improvements Updates the installation script in the component lifecycle. Your core devices must now have Python 3.6 or later, including `pip` for your version of Python, installed on the device before you deploy this component.
1.0.1	Version updated for Greengrass nucleus version 2.3.0 release.
1.0.0	Initial version.

DLR image classification

The DLR image classification component (`aws.greengrass.DLRImageClassification`) contains sample inference code to perform image classification inference using Deep Learning Runtime and resnet-50 models. This component uses the variant DLR image classification model store and the DLR runtime components as dependencies to download DLR and the sample models.

To use this inference component with a custom-trained DLR model, create a custom version of the dependent model store component. To use your own custom inference code, you can use the recipe of this component as a template to create a custom inference component.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Local log file
- Changelog

Versions

This component has the following versions:

- 2.1.x
- 2.0.x

Type

This component is a generic component (`aws.greengrass.generic`). The Greengrass nucleus runs the component's lifecycle scripts.

For more information, see Component types.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

Requirements

This component has the following requirements:

- On Greengrass core devices running Amazon Linux 2 or Ubuntu 18.04, GNU C Library (glibc) version 2.27 or later installed on the device.
- On Armv7l devices, such as Raspberry Pi, dependencies for OpenCV-Python installed on the device. Run the following command to install the dependencies.

```
sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev
 libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
```

- Raspberry Pi devices that run Raspberry Pi OS Bullseye must meet the following requirements:
  - NumPy 1.22.4 or later installed on the device. Raspberry Pi OS Bullseye includes an earlier version of NumPy, so you can run the following command to upgrade NumPy on the device.

    ```
    pip3 install --upgrade numpy
    ```

  - The legacy camera stack enabled on the device. Raspberry Pi OS Bullseye includes a new camera stack that is enabled by default and isn't compatible, so you must enable the legacy camera stack.

    To enable the legacy camera stack

    1. Run the following command to open the Raspberry Pi configuration tool.

       ```
       sudo raspi-config
       ```

    2. Select **Interface Options**.
    3. Select **Legacy camera** to enable the legacy camera stack.
    4. Reboot the Raspberry Pi.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

2.1.13 and 2.1.14

The following table lists the dependencies for version 2.1.13 and 2.1.14 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.13.0	Soft
DLR image classification model store	~2.1.0	Hard
DLR	~1.6.0	Hard

2.1.12

The following table lists the dependencies for version 2.1.12 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.12.0	Soft
DLR image classification model store	~2.1.0	Hard
DLR	~1.6.0	Hard

2.1.11

The following table lists the dependencies for version 2.1.11 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.11.0	Soft
DLR image classification model store	~2.1.0	Hard
DLR	~1.6.0	Hard

2.1.10

The following table lists the dependencies for version 2.1.10 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.10.0	Soft
DLR image classification model store	~2.1.0	Hard
DLR	~1.6.0	Hard

2.1.9

The following table lists the dependencies for version 2.1.9 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.9.0	Soft
DLR image classification model store	~2.1.0	Hard
DLR	~1.6.0	Hard

2.1.8

The following table lists the dependencies for version 2.1.8 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.8.0	Soft
DLR image classification model store	~2.1.0	Hard
DLR	~1.6.0	Hard

2.1.7

The following table lists the dependencies for version 2.1.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.7.0	Soft
DLR image classification model store	~2.1.0	Hard
DLR	~1.6.0	Hard

2.1.6

The following table lists the dependencies for version 2.1.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.6.0	Soft
DLR image classification model store	~2.1.0	Hard
DLR	~1.6.0	Hard

2.1.4 - 2.1.5

The following table lists the dependencies for versions 2.1.4 to 2.1.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.5.0	Soft
DLR image classification model store	~2.1.0	Hard
DLR	~1.6.0	Hard

2.1.3

The following table lists the dependencies for version 2.1.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.4.0	Soft
DLR image classification model store	~2.1.0	Hard
DLR	~1.6.0	Hard

2.1.2

The following table lists the dependencies for version 2.1.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.3.0	Soft
DLR image classification model store	~2.1.0	Hard
DLR	~1.6.0	Hard

2.1.1

The following table lists the dependencies for version 2.1.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.2.0	Soft
DLR image classification model store	~2.1.0	Hard
DLR	~1.6.0	Hard

2.0.x

The following table lists the dependencies for version 2.0.x of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	~2.0.0	Soft
DLR image classification model store	~2.0.0	Hard
DLR	~1.3.0	Soft

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

2.1.x

`accessControl`

(Optional) The object that contains the authorization policy that allows the component to publish messages to the default notifications topic.

Default:

```
{
   "aws.greengrass.ipc.mqttproxy": {
      "aws.greengrass.DLRImageClassification:mqttproxy:1": {
```

```
            "policyDescription": "Allows access to publish via topic ml/dlr/image-
classification.",
            "operations": [
               "aws.greengrass#PublishToIoTCore"
            ],
            "resources": [
               "ml/dlr/image-classification"
            ]
         }
      }
}
```

`PublishResultsOnTopic`

(Optional) The topic on which you want to publish the inference results. If you modify this value, then you must also modify the value of `resources` in the `accessControl` parameter to match your custom topic name.

Default: `ml/dlr/image-classification`

`Accelerator`

The accelerator that you want to use. Supported values are `cpu` and `gpu`.

The sample models in the dependent model component support only CPU acceleration. To use GPU acceleration with a different custom model, create a custom model component to override the public model component.

Default: `cpu`

`ImageDirectory`

(Optional) The path of the folder on the device where inference components read images. You can modify this value to any location on your device to which you have read/write access.

Default: */greengrass/v2*`/packages/artifacts-unarchived/`*component-name*`/image_classification/sample_images/`

> **Note**
>
> If you set the value of `UseCamera` to `true`, then this configuration parameter is ignored.

`ImageName`

(Optional) The name of the image that the inference component uses as an input to a make prediction. The component looks for the image in the folder specified in `ImageDirectory`. By default, the component uses the sample image in the default image directory. AWS IoT Greengrass supports the following image formats: `jpeg`, `jpg`, `png`, and `npy`.

Default: `cat.jpeg`

> **Note**
>
> If you set the value of `UseCamera` to `true`, then this configuration parameter is ignored.

`InferenceInterval`

(Optional) The time in seconds between each prediction made by the inference code. The sample inference code runs indefinitely and repeats its predictions at the specified time interval. For example, you can change this to a shorter interval if you want to use images taken by a camera for real-time prediction.

Default: `3600`

`ModelResourceKey`

(Optional) The models that are used in the dependent public model component. Modify this parameter only if you override the public model component with a custom component.

Default:

```
{
    "armv7l": "DLR-resnet50-armv7l-cpu-ImageClassification",
    "aarch64": "DLR-resnet50-aarch64-cpu-ImageClassification",
    "x86_64": "DLR-resnet50-x86_64-cpu-ImageClassification",
    "windows": "DLR-resnet50-win-cpu-ImageClassification"
}
```

`UseCamera`

(Optional) String value that defines whether to use images from a camera connected to the Greengrass core device. Supported values are `true` and `false`.

When you set this value to `true`, the sample inference code accesses the camera on your device and runs inference locally on the captured image. The values of the `ImageName` and `ImageDirectory` parameters are ignored. Make sure that the user running this component has read/write access to the location where the camera stores captured images.

Default: `false`

> **Note**
>
> When you view the recipe of this component, the `UseCamera` configuration parameter doesn't appear in the default configuration. However, you can modify the value of this parameter in a configuration merge update when you deploy the component.
> When you set `UseCamera` to `true`, you must also create a symlink to enable the inference component to access your camera from the virtual environment that is created by the runtime component. For more information about using a camera with the sample inference components, see Update component configurations.

2.0.x

`MLRootPath`

(Optional) The path of the folder on Linux core devices where inference components read images and write inference results. You can modify this value to any location on your device to which the user running this component has read/write access.

Default: `/greengrass/v2/work/variant.DLR/greengrass_ml`

Default: `/greengrass/v2/work/variant.TensorFlowLite/greengrass_ml`

`Accelerator`

The accelerator that you want to use. Supported values are `cpu` and `gpu`.

The sample models in the dependent model component support only CPU acceleration. To use GPU acceleration with a different custom model, create a custom model component to override the public model component.

Default: `cpu`

`ImageName`

(Optional) The name of the image that the inference component uses as an input to a make prediction. The component looks for the image in the folder specified in `ImageDirectory`. The default location is *`MLRootPath`*`/images`. AWS IoT Greengrass supports the following image formats: `jpeg`, `jpg`, png, and npy.

Default: `cat.jpeg`

`InferenceInterval`

(Optional) The time in seconds between each prediction made by the inference code. The sample inference code runs indefinitely and repeats its predictions at the specified time interval. For example, you can change this to a shorter interval if you want to use images taken by a camera for real-time prediction.

Default: `3600`

`ModelResourceKey`

(Optional) The models that are used in the dependent public model component. Modify this parameter only if you override the public model component with a custom component.

Default:

```
armv7l: "DLR-resnet50-armv7l-cpu-ImageClassification"
x86_64: "DLR-resnet50-x86_64-cpu-ImageClassification"
```

Local log file

This component uses the following log file.

Linux

```
/greengrass/v2/logs/aws.greengrass.DLRImageClassification.log
```

Windows

```
C:\greengrass\v2\logs\aws.greengrass.DLRImageClassification.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/aws.greengrass.DLRImageClassification.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\aws.greengrass.DLRImageClassification.log -
  Tail 10 -Wait
  ```

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.1.14	Version updated for Greengrass nucleus 2.12.5 release.
2.1.13	Version updated for Greengrass nucleus version 2.12.0 release.
2.1.12	Version updated for Greengrass nucleus version 2.11.0 release.
2.1.11	Version updated for Greengrass nucleus version 2.10.0 release.
2.1.10	Version updated for Greengrass nucleus version 2.9.0 release.
2.1.9	Version updated for Greengrass nucleus version 2.8.0 release.
2.1.8	Version updated for Greengrass nucleus version 2.7.0 release.
2.1.7	Version updated for Greengrass nucleus version 2.6.0 release.
2.1.6	Version updated for Greengrass nucleus version 2.5.0 release.
2.1.5	Component released in all AWS Regions.

Version	Changes
2.1.4	Version updated for Greengrass nucleus version 2.4.0 release. This version isn't available in Europe (London) (`eu-west-2`).
2.1.3	Version updated for Greengrass nucleus version 2.3.0 release.
2.1.2	Version updated for Greengrass nucleus version 2.2.0 release.
2.1.1	New features • Use Deep Learning Runtime v1.6.0. • Add support for sample image classification on Armv8 (AArch64) platforms. This extends machine learning support for Greengrass core devices running NVIDIA Jetson, such as the Jetson Nano. • Enable camera integration for sample inference. Use the new `UseCamera` configuration parameter to enable the sample inference code to access the camera on your Greengrass core device and run inference locally on the captured image. • Add support for publishing inference results to the AWS Cloud. Use the new `PublishResultsOnTopic` configuration parameter to specify the topic on which you want to publish results. • Add the new `ImageDirectory` configuration parameter that enables you to specify a custom directory for the image on which you want to perform inference. Bug fixes and improvements • Write inference results to the component log file instead of a separate inference file. • Use the AWS IoT Greengrass Core software logging module to log component output. • Use the AWS IoT Device SDK to read the component configuration and apply configuration changes.
2.0.4	Initial version.

DLR object detection

The DLR object detection component (`aws.greengrass.DLRObjectDetection`) contains sample inference code to perform object detection inference using Deep Learning Runtime and sample pre-trained models. This component uses the variant DLR object detection model store and the DLR runtime components as dependencies to download DLR and the sample models.

To use this inference component with a custom-trained DLR model, create a custom version of the dependent model store component. To use your own custom inference code, you can use the recipe of this component as a template to create a custom inference component.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Local log file
- Changelog

Versions

This component has the following versions:

- 2.1.x
- 2.0.x

Type

This component is a generic component (`aws.greengrass.generic`). The Greengrass nucleus runs the component's lifecycle scripts.

For more information, see Component types.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

Requirements

This component has the following requirements:

- On Greengrass core devices running Amazon Linux 2 or Ubuntu 18.04, GNU C Library (glibc) version 2.27 or later installed on the device.
- On Armv7l devices, such as Raspberry Pi, dependencies for OpenCV-Python installed on the device. Run the following command to install the dependencies.

```
sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev
 libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
```

- Raspberry Pi devices that run Raspberry Pi OS Bullseye must meet the following requirements:
  - NumPy 1.22.4 or later installed on the device. Raspberry Pi OS Bullseye includes an earlier version of NumPy, so you can run the following command to upgrade NumPy on the device.

    ```
    pip3 install --upgrade numpy
    ```

  - The legacy camera stack enabled on the device. Raspberry Pi OS Bullseye includes a new camera stack that is enabled by default and isn't compatible, so you must enable the legacy camera stack.

    To enable the legacy camera stack

    1. Run the following command to open the Raspberry Pi configuration tool.

       ```
       sudo raspi-config
       ```

    2. Select **Interface Options**.
    3. Select **Legacy camera** to enable the legacy camera stack.
    4. Reboot the Raspberry Pi.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

2.1.13 and 2.1.14

The following table lists the dependencies for version 2.1.13 and 2.1.14 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.13.0	Soft
DLR object detection model store	~2.1.0	Hard
DLR	~1.6.0	Hard

2.1.12

The following table lists the dependencies for version 2.1.12 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.12.0	Soft
DLR object detection model store	~2.1.0	Hard
DLR	~1.6.0	Hard

2.1.11

The following table lists the dependencies for version 2.1.11 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.11.0	Soft
DLR object detection model store	~2.1.0	Hard
DLR	~1.6.0	Hard

2.1.10

The following table lists the dependencies for version 2.1.10 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.10.0	Soft
DLR object detection model store	~2.1.0	Hard
DLR	~1.6.0	Hard

2.1.9

The following table lists the dependencies for version 2.1.9 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.9.0	Soft
DLR object detection model store	~2.1.0	Hard
DLR	~1.6.0	Hard

2.1.8

The following table lists the dependencies for version 2.1.8 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.8.0	Soft
DLR object detection model store	~2.1.0	Hard
DLR	~1.6.0	Hard

2.1.7

The following table lists the dependencies for version 2.1.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.7.0	Soft
DLR object detection model store	~2.1.0	Hard
DLR	~1.6.0	Hard

2.1.6

The following table lists the dependencies for version 2.1.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.6.0	Soft
DLR object detection model store	~2.1.0	Hard
DLR	~1.6.0	Hard

2.1.4 - 2.1.5

The following table lists the dependencies for versions 2.1.4 to 2.1.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.5.0	Soft
DLR object detection model store	~2.1.0	Hard
DLR	~1.6.0	Hard

2.1.3

The following table lists the dependencies for version 2.1.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.4.0	Soft
DLR object detection model store	~2.1.0	Hard
DLR	~1.6.0	Hard

2.1.2

The following table lists the dependencies for version 2.1.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.3.0	Soft
DLR object detection model store	~2.1.0	Hard
DLR	~1.6.0	Hard

2.1.1

The following table lists the dependencies for version 2.1.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.2.0	Soft
DLR object detection model store	~2.1.0	Hard
DLR	~1.6.0	Hard

2.0.x

The following table lists the dependencies for version 2.0.x of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	~2.0.0	Soft
DLR object detection model store	~2.0.0	Hard
DLR	~1.3.0	Soft

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

2.1.x

`accessControl`

(Optional) The object that contains the authorization policy that allows the component to publish messages to the default notifications topic.

Default:

```
{
   "aws.greengrass.ipc.mqttproxy": {
      "aws.greengrass.DLRObjectDetection:mqttproxy:1": {
```

```
            "policyDescription": "Allows access to publish via topic ml/dlr/object-
detection.",
            "operations": [
               "aws.greengrass#PublishToIoTCore"
            ],
            "resources": [
               "ml/dlr/object-detection"
            ]
         }
      }
}
```

`PublishResultsOnTopic`

(Optional) The topic on which you want to publish the inference results. If you modify this value, then you must also modify the value of `resources` in the `accessControl` parameter to match your custom topic name.

Default: `ml/dlr/object-detection`

`Accelerator`

The accelerator that you want to use. Supported values are `cpu` and `gpu`.

The sample models in the dependent model component support only CPU acceleration. To use GPU acceleration with a different custom model, create a custom model component to override the public model component.

Default: `cpu`

`ImageDirectory`

(Optional) The path of the folder on the device where inference components read images. You can modify this value to any location on your device to which you have read/write access.

Default: *`/greengrass/v2`*`/packages/artifacts-unarchived/`*`component-name`*`/object_detection/sample_images/`

> **Note**
>
> If you set the value of `UseCamera` to `true`, then this configuration parameter is ignored.

`ImageName`

(Optional) The name of the image that the inference component uses as an input to a make prediction. The component looks for the image in the folder specified in `ImageDirectory`. By default, the component uses the sample image in the default image directory. AWS IoT Greengrass supports the following image formats: `jpeg`, `jpg`, `png`, and `npy`.

Default: `objects.jpg`

> **Note**
>
> If you set the value of `UseCamera` to `true`, then this configuration parameter is ignored.

`InferenceInterval`

(Optional) The time in seconds between each prediction made by the inference code. The sample inference code runs indefinitely and repeats its predictions at the specified time interval. For example, you can change this to a shorter interval if you want to use images taken by a camera for real-time prediction.

Default: `3600`

`ModelResourceKey`

(Optional) The models that are used in the dependent public model component. Modify this parameter only if you override the public model component with a custom component.

Default:

```
{
    "armv7l": "DLR-yolo3-armv7l-cpu-ObjectDetection",
    "aarch64": "DLR-yolo3-aarch64-gpu-ObjectDetection",
    "x86_64": "DLR-yolo3-x86_64-cpu-ObjectDetection",
    "windows": "DLR-resnet50-win-cpu-ObjectDetection"
}
```

`UseCamera`

(Optional) String value that defines whether to use images from a camera connected to the Greengrass core device. Supported values are `true` and `false`.

When you set this value to `true`, the sample inference code accesses the camera on your device and runs inference locally on the captured image. The values of the `ImageName` and `ImageDirectory` parameters are ignored. Make sure that the user running this component has read/write access to the location where the camera stores captured images.

Default: `false`

> **Note**
>
> When you view the recipe of this component, the `UseCamera` configuration parameter doesn't appear in the default configuration. However, you can modify the value of this parameter in a configuration merge update when you deploy the component.
> When you set `UseCamera` to `true`, you must also create a symlink to enable the inference component to access your camera from the virtual environment that is created by the runtime component. For more information about using a camera with the sample inference components, see Update component configurations.

2.0.x

`MLRootPath`

(Optional) The path of the folder on Linux core devices where inference components read images and write inference results. You can modify this value to any location on your device to which the user running this component has read/write access.

Default: */greengrass/v2*`/work/variant.DLR/greengrass_ml`

Default: */greengrass/v2*`/work/variant.TensorFlowLite/greengrass_ml`

`Accelerator`

Do not modify. Currently, the only supported value for the accelerator is `cpu`, because the models in the dependent model components are compiled only for the CPU accelerator.

`ImageName`

(Optional) The name of the image that the inference component uses as an input to a make prediction. The component looks for the image in the folder specified in `ImageDirectory`.

The default location is *MLRootPath*/images. AWS IoT Greengrass supports the following image formats: jpeg, jpg, png, and npy.

Default: objects.jpg

`InferenceInterval`

(Optional) The time in seconds between each prediction made by the inference code. The sample inference code runs indefinitely and repeats its predictions at the specified time interval. For example, you can change this to a shorter interval if you want to use images taken by a camera for real-time prediction.

Default: 3600

`ModelResourceKey`

(Optional) The models that are used in the dependent public model component. Modify this parameter only if you override the public model component with a custom component.

Default:

```
{
    armv7l: "DLR-yolo3-armv7l-cpu-ObjectDetection",
    x86_64: "DLR-yolo3-x86_64-cpu-ObjectDetection"
}
```

Local log file

This component uses the following log file.

Linux

```
/greengrass/v2/logs/aws.greengrass.DLRObjectDetection.log
```

Windows

```
C:\greengrass\v2\logs\aws.greengrass.DLRObjectDetection.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/aws.greengrass.DLRObjectDetection.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\aws.greengrass.DLRObjectDetection.log -Tail 10
   -Wait
  ```

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.1.14	Version updated for Greengrass nucleus 2.12.5 release.
2.1.13	Version updated for Greengrass nucleus version 2.12.0 release.
2.1.12	Version updated for Greengrass nucleus version 2.11.0 release.
2.1.11	Version updated for Greengrass nucleus version 2.10.0 release.
2.1.10	Version updated for Greengrass nucleus version 2.9.0 release.
2.1.9	Version updated for Greengrass nucleus version 2.8.0 release.
2.1.8	Version updated for Greengrass nucleus version 2.7.0 release.
2.1.7	Version updated for Greengrass nucleus version 2.6.0 release.
2.1.6	Version updated for Greengrass nucleus version 2.5.0 release.
2.1.5	Component released in all AWS Regions.

Version	Changes
2.1.4	Version updated for Greengrass nucleus version 2.4.0 release. This version isn't available in Europe (London) (`eu-west-2`).
2.1.3	Version updated for Greengrass nucleus version 2.3.0 release.
2.1.2	Bug fixes and improvements • Fixes an image scaling issue that resulted in inaccurate bounding boxes in the sample DLR object detection inference results.
2.1.1	New features • Use Deep Learning Runtime v1.6.0. • Add support for sample object detection on Armv8 (AArch64) platforms. This extends machine learning support for Greengrass core devices running NVIDIA Jetson, such as the Jetson Nano. • Enable camera integration for sample inference. Use the new `UseCamera` configuration parameter to enable the sample inference code to access the camera on your Greengrass core device and run inference locally on the captured image. • Add support for publishing inference results to the AWS Cloud. Use the new `PublishResultsOnTopic` configuration parameter to specify the topic on which you want to publish results. • Add the new `ImageDirectory` configuration parameter that enables you to specify a custom directory for the image on which you want to perform inference. Bug fixes and improvements • Write inference results to the component log file instead of a separate inference file. • Use the AWS IoT Greengrass Core software logging module to log component output. • Use the AWS IoT Device SDK to read the component configuration and apply configuration changes.
2.0.4	Initial version.

DLR image classification model store

The DLR image classification model store is a machine learning model component that contains pre-trained ResNet-50 models as Greengrass artifacts. The pre-trained models used in this component are fetched from the GluonCV Model Zoo and are compiled using SageMaker AI Neo Deep Learning Runtime.

The DLR image classification inference component uses this component as a dependency for the model source. To use a custom-trained DLR model, create a custom version of this model component, and include your custom model as a component artifact. You can use the recipe of this component as a template to create custom model components.

> **Note**
>
> The name of the DLR image classification model store component varies depending on its version. The component name for version 2.1.x and later versions is `variant.DLR.ImageClassification.ModelStore`. The component name for version 2.0.x is `variant.ImageClassification.ModelStore`.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Local log file
- Changelog

Versions

This component has the following versions:

- 2.1.x (`variant.DLR.ImageClassification.ModelStore`)
- 2.0.x (`variant.ImageClassification.ModelStore`)

Type

This component is a generic component (`aws.greengrass.generic`). The Greengrass nucleus runs the component's lifecycle scripts.

For more information, see Component types.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

Requirements

This component has the following requirements:

- On Greengrass core devices running Amazon Linux 2 or Ubuntu 18.04, GNU C Library (glibc) version 2.27 or later installed on the device.
- On Armv7l devices, such as Raspberry Pi, dependencies for OpenCV-Python installed on the device. Run the following command to install the dependencies.

  ```
  sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev
   libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
  ```

- Raspberry Pi devices that run Raspberry Pi OS Bullseye must meet the following requirements:
  - NumPy 1.22.4 or later installed on the device. Raspberry Pi OS Bullseye includes an earlier version of NumPy, so you can run the following command to upgrade NumPy on the device.

    ```
    pip3 install --upgrade numpy
    ```

  - The legacy camera stack enabled on the device. Raspberry Pi OS Bullseye includes a new camera stack that is enabled by default and isn't compatible, so you must enable the legacy camera stack.

    To enable the legacy camera stack

    1. Run the following command to open the Raspberry Pi configuration tool.

```
sudo raspi-config
```

2. Select **Interface Options**.
3. Select **Legacy camera** to enable the legacy camera stack.
4. Reboot the Raspberry Pi.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

2.1.12 - 2.1.14

The following table lists the dependencies for version 2.1.12 and 2.1.13 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.13.0	Soft

2.1.11

The following table lists the dependencies for version 2.1.11 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.12.0	Soft

2.1.10

The following table lists the dependencies for version 2.1.10 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.11.0	Soft

2.1.9

The following table lists the dependencies for version 2.1.9 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.10.0	Soft

2.1.8

The following table lists the dependencies for version 2.1.8 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.9.0	Soft

2.1.7

The following table lists the dependencies for version 2.1.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.8.0	Soft

2.1.6

The following table lists the dependencies for version 2.1.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.7.0	Soft

2.1.5

The following table lists the dependencies for version 2.1.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.6.0	Soft

2.1.4

The following table lists the dependencies for version 2.1.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.5.0	Soft

2.1.3

The following table lists the dependencies for version 2.1.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.4.0	Soft

2.1.2

The following table lists the dependencies for version 2.1.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.3.0	Soft

2.1.1

The following table lists the dependencies for version 2.1.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.2.0	Soft

2.0.x

The following table lists the dependencies for version 2.0.x of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	~2.0.0	Soft

Configuration

This component doesn't have any configuration parameters.

Local log file

This component doesn't output logs.

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.1.13	Version updated for Greengrass nucleus 2.12.5 release.
2.1.12	Version updated for Greengrass nucleus version 2.12.0 release.
2.1.11	Version updated for Greengrass nucleus version 2.11.0 release.
2.1.10	Version updated for Greengrass nucleus version 2.10.0 release.
2.1.9	Version updated for Greengrass nucleus version 2.9.0 release.
2.1.8	Version updated for Greengrass nucleus version 2.8.0 release.
2.1.7	Version updated for Greengrass nucleus version 2.7.0 release.

Version	Changes
2.1.6	Version updated for Greengrass nucleus version 2.6.0 release.
2.1.5	New features • Adds sample image classification models for Windows core devices. • Version updated for Greengrass nucleus version 2.5.0 release.
2.1.4	Version updated for Greengrass nucleus version 2.4.0 release.
2.1.3	Version updated for Greengrass nucleus version 2.3.0 release.
2.1.2	Version updated for Greengrass nucleus version 2.2.0 release.
2.1.1	New features • Add a sample ResNet-50 image classification model for Armv8 (AArch64) platforms. This extends machine learning support for Greengrass core devices running NVIDIA Jetson, such as the Jetson Nano.
2.0.4	Initial version.

DLR object detection model store

The DLR object detection model store is a machine learning model component that contains pre-trained YOLOv3 models as Greengrass artifacts. The sample models used in this component are fetched from the GluonCV Model Zoo and compiled using SageMaker AI Neo Deep Learning Runtime.

The DLR object detection inference component uses this component as a dependency for the model source. To use a custom-trained DLR model, create a custom version of this model component, and include your custom model as a component artifact. You can use the recipe of this component as a template to create custom model components.

> **Note**
>
> The name of the DLR object detection model store component varies depending on its version. The component name for version 2.1.x and later versions is

`variant.DLR.ObjectDetection.ModelStore`. The component name for version 2.0.x is `variant.ObjectDetection.ModelStore`.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Local log file
- Changelog

Versions

This component has the following versions:

- 2.1.x
- 2.0.x

Type

This component is a generic component (`aws.greengrass.generic`). The Greengrass nucleus runs the component's lifecycle scripts.

For more information, see Component types.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

Requirements

This component has the following requirements:

- On Greengrass core devices running Amazon Linux 2 or Ubuntu 18.04, GNU C Library (glibc) version 2.27 or later installed on the device.
- On Armv7l devices, such as Raspberry Pi, dependencies for OpenCV-Python installed on the device. Run the following command to install the dependencies.

  ```
  sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev
   libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
  ```
- Raspberry Pi devices that run Raspberry Pi OS Bullseye must meet the following requirements:
  - NumPy 1.22.4 or later installed on the device. Raspberry Pi OS Bullseye includes an earlier version of NumPy, so you can run the following command to upgrade NumPy on the device.

    ```
    pip3 install --upgrade numpy
    ```
  - The legacy camera stack enabled on the device. Raspberry Pi OS Bullseye includes a new camera stack that is enabled by default and isn't compatible, so you must enable the legacy camera stack.

    To enable the legacy camera stack

    1. Run the following command to open the Raspberry Pi configuration tool.

       ```
       sudo raspi-config
       ```
    2. Select **Interface Options**.
    3. Select **Legacy camera** to enable the legacy camera stack.
    4. Reboot the Raspberry Pi.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of

the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

2.1.13 and 2.1.14

The following table lists the dependencies for version 2.1.13 and 2.1.14 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.13.0	Soft

2.1.12

The following table lists the dependencies for version 2.1.12 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.12.0	Soft

2.1.11

The following table lists the dependencies for version 2.1.11 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.11.0	Soft

2.1.10

The following table lists the dependencies for version 2.1.10 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.10.0	Soft

2.1.9

The following table lists the dependencies for version 2.1.9 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.9.0	Soft

2.1.8

The following table lists the dependencies for version 2.1.8 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.8.0	Soft

2.1.7

The following table lists the dependencies for version 2.1.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.7.0	Soft

2.1.5 and 2.1.6

The following table lists the dependencies for versions 2.1.5 and 2.1.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.6.0	Soft

2.1.4

The following table lists the dependencies for version 2.1.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.5.0	Soft

2.1.3

The following table lists the dependencies for version 2.1.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.4.0	Soft

2.1.2

The following table lists the dependencies for version 2.1.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.3.0	Soft

2.1.1

The following table lists the dependencies for version 2.1.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.2.0	Soft

2.0.x

The following table lists the dependencies for version 2.0.x of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	~2.0.0	Soft

Configuration

This component doesn't have any configuration parameters.

Local log file

This component doesn't output logs.

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.1.14	Version updated for Greengrass nucleus 2.12.5 release.
2.1.13	Version updated for Greengrass nucleus version 2.12.0 release.
2.1.12	Version updated for Greengrass nucleus version 2.11.0 release.
2.1.11	Version updated for Greengrass nucleus version 2.10.0 release.
2.1.10	Version updated for Greengrass nucleus version 2.9.0 release.
2.1.9	Version updated for Greengrass nucleus version 2.8.0 release.
2.1.8	Version updated for Greengrass nucleus version 2.7.0 release.
2.1.7	Version updated for Greengrass nucleus version 2.6.0 release.
2.1.6	Adds a CPU model to fix an issue on Armv8 (AArch64) devices.
2.1.5	New features • Adds sample object detection models for Windows core devices. Bug fixes and improvements • Version updated for Greengrass nucleus version 2.5.0 release.
2.1.4	Version updated for Greengrass nucleus version 2.4.0 release.
2.1.3	Version updated for Greengrass nucleus version 2.3.0 release.
2.1.2	Version updated for Greengrass nucleus version 2.2.0 release.

Version	Changes
2.1.1	New features • Add a sample YOLOv3 object detection model for Armv8 (AArch64) platforms. This extends machine learning support for Greengrass core devices running NVIDIA Jetson, such as the Jetson Nano.
2.0.4	Initial version.

DLR runtime

The DLR runtime component (`variant.DLR`) contains a script that installs Deep Learning Runtime (DLR) and its dependencies in a virtual environment on your device. The DLR image classification and DLR object detection components use this component as a dependency for installing DLR. Component version 1.6.x installs DLR v1.6.0 and component version 1.3.x installs DLR v1.3.0.

To use a different runtime, you can use the recipe of this component as a template to create a custom machine learning component.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Usage
- Local log file
- Changelog

Versions

This component has the following versions:

- 1.6.x

- 1.3.x

Type

This component is a generic component (`aws.greengrass.generic`). The Greengrass nucleus runs the component's lifecycle scripts.

For more information, see Component types.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

Requirements

This component has the following requirements:

- On Greengrass core devices running Amazon Linux 2 or Ubuntu 18.04, GNU C Library (glibc) version 2.27 or later installed on the device.
- On Armv7l devices, such as Raspberry Pi, dependencies for OpenCV-Python installed on the device. Run the following command to install the dependencies.

  ```
  sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev
   libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
  ```

- Raspberry Pi devices that run Raspberry Pi OS Bullseye must meet the following requirements:
  - NumPy 1.22.4 or later installed on the device. Raspberry Pi OS Bullseye includes an earlier version of NumPy, so you can run the following command to upgrade NumPy on the device.

    ```
    pip3 install --upgrade numpy
    ```

  - The legacy camera stack enabled on the device. Raspberry Pi OS Bullseye includes a new camera stack that is enabled by default and isn't compatible, so you must enable the legacy camera stack.

To enable the legacy camera stack

1. Run the following command to open the Raspberry Pi configuration tool.

   ```
   sudo raspi-config
   ```

2. Select **Interface Options**.
3. Select **Legacy camera** to enable the legacy camera stack.
4. Reboot the Raspberry Pi.

Endpoints and ports

By default, this component uses an installer script to install packages using the `apt`, `yum`, `brew`, and `pip` commands, depending on what platform the core device uses. This component must be able to perform outbound requests to various package indexes and repositories to run the installer script. To allow this component's outbound traffic through a proxy or firewall, you must identify the endpoints for the package indexes and repositories where your core device connects to install.

Consider the following when you identify endpoints required for this component's install script:

- The endpoints depend on the core device's platform. For example, a core device that runs Ubuntu uses `apt` rather than `yum` or `brew`. Additionally, devices that use the same package index might have different source lists, so they might retrieve packages from different repositories.
- The endpoints might differ between multiple devices that use the same package index, because each device has its own source lists that define where to retrieve packages.
- The endpoints might change over time. Each package index provides the URLs of the repositories where you download packages, and the owner of a package can change what URLs the package index provides.

For more information about the dependencies that this component installs, and how to disable the installer script, see the UseInstaller configuration parameter.

For more information about endpoints and ports required for basic operation, see Allow device traffic through a proxy or firewall.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

1.6.11 - 1.6.16

The following table lists the dependencies for versions 1.6.11 to 1.6.16 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <3.0.0	Soft

1.6.10

The following table lists the dependencies for version 1.6.10 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.9.0	Soft

1.6.9

The following table lists the dependencies for version 1.6.9 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.8.0	Soft

1.6.8

The following table lists the dependencies for version 1.6.8 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.7.0	Soft

1.6.6 and 1.6.7

The following table lists the dependencies for versions 1.6.6 and 1.6.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.6.0	Soft

1.6.4 and 1.6.5

The following table lists the dependencies for versions 1.6.4 and 1.6.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.5.0	Soft

1.6.3

The following table lists the dependencies for version 1.6.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.4.0	Soft

1.6.2

The following table lists the dependencies for version 1.6.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.3.0	Soft

1.6.1

The following table lists the dependencies for version 1.6.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.2.0	Soft

1.3.x

The following table lists the dependencies for version 1.3.x of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	~2.0.0	Soft

For more information about component dependencies, see the component recipe reference.

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

`MLRootPath`

(Optional) The path of the folder on Linux core devices where inference components read images and write inference results. You can modify this value to any location on your device to which the user running this component has read/write access.

Default: `/greengrass/v2/work/variant.DLR/greengrass_ml`

`WindowsMLRootPath`

This feature is available in v1.6.6 and later of this component.

(Optional) The path of the folder on Windows core device where inference components read images and write inference results. You can modify this value to any location on your device to which the user running this component has read/write access.

Default: `C:\greengrass\v2\\work\\variant.DLR\\greengrass_ml`

`UseInstaller`

(Optional) String value that defines whether to use the installer script in this component to install DLR and its dependencies. Supported values are `true` and `false`.

Set this value to `false` if you want to use a custom script for DLR installation, or if you want to include runtime dependencies in a pre-built Linux image. To use this component with the AWS-provided DLR inference components, install the following libraries, including any dependencies, and make them available to the system user, such as `ggc_user`, that runs the ML components.

- Python 3.7 or later, including `pip` for your version of Python.
- Deep Learning Runtime v1.6.0
- NumPy.
- OpenCV-Python.
- AWS IoT Device SDK v2 for Python.
- AWS Common Runtime (CRT) Python.
- Picamera (for Raspberry Pi devices only).
- `awscam` module (for AWS DeepLens devices).
- libGL (for Linux devices)

Default: `true`

Usage

Use this component with the `UseInstaller` configuration parameter set to `true` to install DLR and its dependencies on your device. The component sets up a virtual environment on your device that includes the OpenCV and NumPy libraries that are required for DLR.

> **Note**
>
> The installer script in this component also installs the latest versions of additional system libraries that are required to configure the virtual environment on your device and to use the installed machine learning framework. This might upgrade the existing system libraries on your device. Review the following table for the list of libraries that this component installs for each supported operating system. If you want to customize this installation process, set the `UseInstaller` configuration parameter to `false`, and develop your own installer script.

Platform	Libraries installed on the device system	Libraries installed in the virtual environment
Armv7l	`build-essential` , `cmake`, `ca-certificates` , `git`	`setuptools` , `wheel`
Amazon Linux 2	`mesa-libGL`	None
Ubuntu	`wget`	None

When you deploy your inference component, this runtime component first verifies if your device already has DLR and its dependencies installed, and if not, then it installs them for you.

Local log file

This component uses the following log file.

Linux

```
/greengrass/v2/logs/variant.DLR.log
```

Windows

```
C:\greengrass\v2\logs\variant.DLR.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/variant.DLR.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\variant.DLR.log -Tail 10 -Wait
  ```

Changelog

The following table describes the changes in each version of the component.

Version	Changes
1.6.16	Version updated for Greengrass nucleus version 2.12.5.
1.6.12	Bug fixes and improvements • Fixes the installation script for Windows OS users.
1.6.11	Version updated for Greengrass nucleus version 2.9.0 release.
1.6.10	Version updated for Greengrass nucleus version 2.8.0 release.
1.6.9	Version updated for Greengrass nucleus version 2.7.0 release.
1.6.8	Version updated for Greengrass nucleus version 2.6.0 release.
1.6.7	Bug fixes and improvements • Updates the `UseInstaller` installation script to install libGL, which isn't available by default on certain Linux platforms. • Updates the `UseInstaller` installation script to always use Python 3.9 in this component's virtual environment. This change helps ensure compatibility with other libraries.
1.6.6	New features • Adds support for core devices that run Windows. • Adds the new `WindowsMLRootPath` configuration parameter that you can use to configure the inference results folder on Windows core devices.
1.6.5	New features • Adds the new `UseInstaller` configuration parameter that you can use to disable the installation script in this component.
1.6.4	Version updated for Greengrass nucleus version 2.4.0 release.
1.6.3	Version updated for Greengrass nucleus version 2.3.0 release.

Version	Changes
1.6.2	Version updated for Greengrass nucleus version 2.2.0 release.
1.6.1	New features • Install Deep Learning Runtime v1.6.0 and its dependencies. • Add support for installing DLR on Armv8 (AArch64) platforms. This extends machine learning support for Greengrass core devices running NVIDIA Jetson, such as the Jetson Nano. Bug fixes and improvements • Install the AWS IoT Device SDK in the virtual environment to read the component configuration and apply configuration changes. • Additional minor bug fixes and improvements.
1.3.2	Initial version. Installs DLR v1.3.0.

TensorFlow Lite image classification

The TensorFlow Lite image classification component (`aws.greengrass.TensorFlowLiteImageClassification`) contains sample inference code to perform image classification inference using the TensorFlow Lite runtime and a sample pre-trained MobileNet 1.0 quantized model. This component uses the variant TensorFlow Lite image classification model store and the TensorFlow Lite runtime components as dependencies to download the TensorFlow Lite runtime and the sample model.

To use this inference component with a custom-trained TensorFlow Lite model, create a custom version of the dependent model store component. To use your own custom inference code, you can use the recipe of this component as a template to create a custom inference component.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies

- Configuration
- Local log file
- Changelog

Versions

This component has the following versions:

- 2.1.x

Type

This component is a generic component (`aws.greengrass.generic`). The Greengrass nucleus runs the component's lifecycle scripts.

For more information, see Component types.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

Requirements

This component has the following requirements:

- On Greengrass core devices running Amazon Linux 2 or Ubuntu 18.04, GNU C Library (glibc) version 2.27 or later installed on the device.
- On Armv7l devices, such as Raspberry Pi, dependencies for OpenCV-Python installed on the device. Run the following command to install the dependencies.

  ```
  sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev
   libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
  ```

- Raspberry Pi devices that run Raspberry Pi OS Bullseye must meet the following requirements:
  - NumPy 1.22.4 or later installed on the device. Raspberry Pi OS Bullseye includes an earlier version of NumPy, so you can run the following command to upgrade NumPy on the device.

```
pip3 install --upgrade numpy
```

- The legacy camera stack enabled on the device. Raspberry Pi OS Bullseye includes a new camera stack that is enabled by default and isn't compatible, so you must enable the legacy camera stack.

  To enable the legacy camera stack

  1. Run the following command to open the Raspberry Pi configuration tool.

     ```
     sudo raspi-config
     ```

  2. Select **Interface Options**.
  3. Select **Legacy camera** to enable the legacy camera stack.
  4. Reboot the Raspberry Pi.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

2.1.11 and 2.1.12

The following table lists the dependencies for version 2.1.11 and 2.1.12 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.13.0	Soft
TensorFlow Lite image classification model store	>=2.1.0 <2.2.0	Hard
TensorFlow Lite	>=2.5.0 <2.6.0	Hard

2.1.10

The following table lists the dependencies for version 2.1.10 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.12.0	Soft
TensorFlow Lite image classification model store	>=2.1.0 <2.2.0	Hard
TensorFlow Lite	>=2.5.0 <2.6.0	Hard

2.1.9

The following table lists the dependencies for version 2.1.9 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.11.0	Soft
TensorFlow Lite image classification model store	>=2.1.0 <2.2.0	Hard
TensorFlow Lite	>=2.5.0 <2.6.0	Hard

2.1.8

The following table lists the dependencies for version 2.1.8 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.10.0	Soft
TensorFlow Lite image classification model store	>=2.1.0 <2.2.0	Hard
TensorFlow Lite	>=2.5.0 <2.6.0	Hard

2.1.7

The following table lists the dependencies for version 2.1.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.9.0	Soft
TensorFlow Lite image classification model store	>=2.1.0 <2.2.0	Hard
TensorFlow Lite	>=2.5.0 <2.6.0	Hard

2.1.6

The following table lists the dependencies for version 2.1.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.8.0	Soft
TensorFlow Lite image classification model store	>=2.1.0 <2.2.0	Hard
TensorFlow Lite	>=2.5.0 <2.6.0	Hard

2.1.5

The following table lists the dependencies for version 2.1.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.7.0	Soft
TensorFlow Lite image classification model store	>=2.1.0 <2.2.0	Hard
TensorFlow Lite	>=2.5.0 <2.6.0	Hard

2.1.4

The following table lists the dependencies for version 2.1.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.6.0	Soft
TensorFlow Lite image classification model store	>=2.1.0 <2.2.0	Hard
TensorFlow Lite	>=2.5.0 <2.6.0	Hard

2.1.3

The following table lists the dependencies for version 2.1.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.5.0	Soft
TensorFlow Lite image classification model store	>=2.1.0 <2.2.0	Hard
TensorFlow Lite	>=2.5.0 <2.6.0	Hard

2.1.2

The following table lists the dependencies for version 2.1.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.4.0	Soft
TensorFlow Lite image classification model store	>=2.1.0 <2.2.0	Hard
TensorFlow Lite	>=2.5.0 <2.6.0	Hard

2.1.1

The following table lists the dependencies for version 2.1.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.3.0	Soft
TensorFlow Lite image classification model store	>=2.1.0 <2.2.0	Hard
TensorFlow Lite	>=2.5.0 <2.6.0	Hard

2.1.0

The following table lists the dependencies for version 2.1.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.2.0	Soft
TensorFlow Lite image classification model store	>=2.1.0 <2.2.0	Hard
TensorFlow Lite	>=2.5.0 <2.6.0	Hard

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

`accessControl`

(Optional) The object that contains the authorization policy that allows the component to publish messages to the default notifications topic.

Default:

```
{
   "aws.greengrass.ipc.mqttproxy": {
```

```
      "aws.greengrass.TensorFlowLiteImageClassification:mqttproxy:1": {
         "policyDescription": "Allows access to publish via topic ml/tflite/image-
classification.",
         "operations": [
            "aws.greengrass#PublishToIoTCore"
         ],
         "resources": [
            "ml/tflite/image-classification"
         ]
      }
   }
}
```

`PublishResultsOnTopic`

(Optional) The topic on which you want to publish the inference results. If you modify this value, then you must also modify the value of `resources` in the `accessControl` parameter to match your custom topic name.

Default: `ml/tflite/image-classification`

`Accelerator`

The accelerator that you want to use. Supported values are `cpu` and `gpu`.

The sample models in the dependent model component support only CPU acceleration. To use GPU acceleration with a different custom model, create a custom model component to override the public model component.

Default: `cpu`

`ImageDirectory`

(Optional) The path of the folder on the device where inference components read images. You can modify this value to any location on your device to which you have read/write access.

Default: *`/greengrass/v2`*`/packages/artifacts-unarchived/`*`component-name`*`/image_classification/sample_images/`

> **Note**
>
> If you set the value of `UseCamera` to `true`, then this configuration parameter is ignored.

`ImageName`

(Optional) The name of the image that the inference component uses as an input to a make prediction. The component looks for the image in the folder specified in `ImageDirectory`. By default, the component uses the sample image in the default image directory. AWS IoT Greengrass supports the following image formats: `jpeg`, `jpg`, `png`, and `npy`.

Default: `cat.jpeg`

> **Note**
>
> If you set the value of `UseCamera` to `true`, then this configuration parameter is ignored.

`InferenceInterval`

(Optional) The time in seconds between each prediction made by the inference code. The sample inference code runs indefinitely and repeats its predictions at the specified time interval. For example, you can change this to a shorter interval if you want to use images taken by a camera for real-time prediction.

Default: `3600`

`ModelResourceKey`

(Optional) The models that are used in the dependent public model component. Modify this parameter only if you override the public model component with a custom component.

Default:

```
{
    "model": "TensorFlowLite-Mobilenet"
}
```

`UseCamera`

(Optional) String value that defines whether to use images from a camera connected to the Greengrass core device. Supported values are `true` and `false`.

When you set this value to `true`, the sample inference code accesses the camera on your device and runs inference locally on the captured image. The values of the `ImageName` and

`ImageDirectory` parameters are ignored. Make sure that the user running this component has read/write access to the location where the camera stores captured images.

Default: `false`

> **Note**
>
> When you view the recipe of this component, the `UseCamera` configuration parameter doesn't appear in the default configuration. However, you can modify the value of this parameter in a configuration merge update when you deploy the component.
> When you set `UseCamera` to `true`, you must also create a symlink to enable the inference component to access your camera from the virtual environment that is created by the runtime component. For more information about using a camera with the sample inference components, see Update component configurations.

Local log file

This component uses the following log file.

Linux

```
/greengrass/v2/logs/aws.greengrass.TensorFlowLiteImageClassification.log
```

Windows

```
C:\greengrass\v2\logs\aws.greengrass.TensorFlowLiteImageClassification.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/
  aws.greengrass.TensorFlowLiteImageClassification.log
  ```

Windows (PowerShell)

```
Get-Content C:\greengrass\v2\logs
\aws.greengrass.TensorFlowLiteImageClassification.log -Tail 10 -Wait
```

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.1.12	Version updated for Greengrass nucleus 2.12.5 release.
2.1.11	Version updated for Greengrass nucleus version 2.12.0 release.
2.1.10	Version updated for Greengrass nucleus version 2.11.0 release.
2.1.9	Version updated for Greengrass nucleus version 2.10.0 release.
2.1.8	Version updated for Greengrass nucleus version 2.9.0 release.
2.1.7	Version updated for Greengrass nucleus version 2.8.0 release.
2.1.6	Version updated for Greengrass nucleus version 2.7.0 release.
2.1.5	Version updated for Greengrass nucleus version 2.6.0 release.
2.1.4	Version updated for Greengrass nucleus version 2.5.0 release.
2.1.3	Version updated for Greengrass nucleus version 2.4.0 release.
2.1.2	Version updated for Greengrass nucleus version 2.3.0 release.
2.1.1	Version updated for Greengrass nucleus version 2.2.0 release.
2.1.0	Initial version.

TensorFlow Lite object detection

The TensorFlow Lite object detection component (`aws.greengrass.TensorFlowLiteObjectDetection`) contains sample inference code to perform object detection inference using TensorFlow Lite and a sample pre-trained Single Shot Detection (SSD) MobileNet 1.0 model. This component uses the variant TensorFlow Lite object detection model store and the TensorFlow Lite runtime components as dependencies to download TensorFlow Lite and the sample model.

To use this inference component with a custom-trained TensorFlow Lite model, you can create a custom version of the dependent model store component. To use your own custom inference code, use the recipe of this component as a template to create a custom inference component.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Local log file
- Changelog

Versions

This component has the following versions:

- 2.1.x

Type

This component is a generic component (`aws.greengrass.generic`). The Greengrass nucleus runs the component's lifecycle scripts.

For more information, see Component types.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

Requirements

This component has the following requirements:

- On Greengrass core devices running Amazon Linux 2 or Ubuntu 18.04, GNU C Library (glibc) version 2.27 or later installed on the device.
- On Armv7l devices, such as Raspberry Pi, dependencies for OpenCV-Python installed on the device. Run the following command to install the dependencies.

```
sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev
 libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
```

- Raspberry Pi devices that run Raspberry Pi OS Bullseye must meet the following requirements:
  - NumPy 1.22.4 or later installed on the device. Raspberry Pi OS Bullseye includes an earlier version of NumPy, so you can run the following command to upgrade NumPy on the device.

    ```
    pip3 install --upgrade numpy
    ```

  - The legacy camera stack enabled on the device. Raspberry Pi OS Bullseye includes a new camera stack that is enabled by default and isn't compatible, so you must enable the legacy camera stack.

    To enable the legacy camera stack

    1. Run the following command to open the Raspberry Pi configuration tool.

       ```
       sudo raspi-config
       ```

    2. Select **Interface Options**.
    3. Select **Legacy camera** to enable the legacy camera stack.
    4. Reboot the Raspberry Pi.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

2.1.11 and 2.1.12

The following table lists the dependencies for version 2.1.11 and 2.1.12 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.13.0	Soft
TensorFlow Lite image classification model store	>=2.1.0 <2.2.0	Hard
TensorFlow Lite	>=2.5.0 <2.6.0	Hard

2.1.10

The following table lists the dependencies for version 2.1.10 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.12.0	Soft
TensorFlow Lite image classification model store	>=2.1.0 <2.2.0	Hard
TensorFlow Lite	>=2.5.0 <2.6.0	Hard

2.1.9

The following table lists the dependencies for version 2.1.9 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.11.0	Soft
TensorFlow Lite image classification model store	>=2.1.0 <2.2.0	Hard
TensorFlow Lite	>=2.5.0 <2.6.0	Hard

2.1.8

The following table lists the dependencies for version 2.1.8 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.10.0	Soft
TensorFlow Lite image classification model store	>=2.1.0 <2.2.0	Hard
TensorFlow Lite	>=2.5.0 <2.6.0	Hard

2.1.7

The following table lists the dependencies for version 2.1.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.9.0	Soft
TensorFlow Lite image classification model store	>=2.1.0 <2.2.0	Hard
TensorFlow Lite	>=2.5.0 <2.6.0	Hard

2.1.6

The following table lists the dependencies for version 2.1.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.8.0	Soft
TensorFlow Lite image classification model store	>=2.1.0 <2.2.0	Hard
TensorFlow Lite	>=2.5.0 <2.6.0	Hard

2.1.5

The following table lists the dependencies for version 2.1.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.7.0	Soft
TensorFlow Lite image classification model store	>=2.1.0 <2.2.0	Hard
TensorFlow Lite	>=2.5.0 <2.6.0	Hard

2.1.4

The following table lists the dependencies for version 2.1.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.6.0	Soft
TensorFlow Lite image classification model store	>=2.1.0 <2.2.0	Hard
TensorFlow Lite	>=2.5.0 <2.6.0	Hard

2.1.3

The following table lists the dependencies for version 2.1.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.5.0	Soft
TensorFlow Lite image classification model store	>=2.1.0 <2.2.0	Hard
TensorFlow Lite	>=2.5.0 <2.6.0	Hard

2.1.2

The following table lists the dependencies for version 2.1.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.4.0	Soft
TensorFlow Lite image classification model store	>=2.1.0 <2.2.0	Hard
TensorFlow Lite	>=2.5.0 <2.6.0	Hard

2.1.1

The following table lists the dependencies for version 2.1.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.3.0	Soft
TensorFlow Lite image classification model store	>=2.1.0 <2.2.0	Hard
TensorFlow Lite	>=2.5.0 <2.6.0	Hard

2.1.0

The following table lists the dependencies for version 2.1.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.2.0	Soft
TensorFlow Lite image classification model store	>=2.1.0 <2.2.0	Hard
TensorFlow Lite	>=2.5.0 <2.6.0	Hard

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

`accessControl`

(Optional) The object that contains the authorization policy that allows the component to publish messages to the default notifications topic.

Default:

```
{
   "aws.greengrass.ipc.mqttproxy": {
      "aws.greengrass.TensorFlowLiteObjectDetection:mqttproxy:1": {
         "policyDescription": "Allows access to publish via topic ml/tflite/object-
detection.",
         "operations": [
            "aws.greengrass#PublishToIoTCore"
         ],
         "resources": [
            "ml/tflite/object-detection"
         ]
      }
   }
}
```

`PublishResultsOnTopic`

(Optional) The topic on which you want to publish the inference results. If you modify this value, then you must also modify the value of `resources` in the `accessControl` parameter to match your custom topic name.

Default: `ml/tflite/object-detection`

`Accelerator`

The accelerator that you want to use. Supported values are `cpu` and `gpu`.

The sample models in the dependent model component support only CPU acceleration. To use GPU acceleration with a different custom model, create a custom model component to override the public model component.

Default: `cpu`

`ImageDirectory`

(Optional) The path of the folder on the device where inference components read images. You can modify this value to any location on your device to which you have read/write access.

Default: *`/greengrass/v2`*`/packages/artifacts-unarchived/`*`component-name`*`/object_detection/sample_images/`

> **Note**
>
> If you set the value of `UseCamera` to `true`, then this configuration parameter is ignored.

`ImageName`

(Optional) The name of the image that the inference component uses as an input to a make prediction. The component looks for the image in the folder specified in `ImageDirectory`. By default, the component uses the sample image in the default image directory. AWS IoT Greengrass supports the following image formats: `jpeg`, `jpg`, `png`, and `npy`.

Default: `objects.jpg`

> **Note**
>
> If you set the value of `UseCamera` to `true`, then this configuration parameter is ignored.

`InferenceInterval`

(Optional) The time in seconds between each prediction made by the inference code. The sample inference code runs indefinitely and repeats its predictions at the specified time interval. For example, you can change this to a shorter interval if you want to use images taken by a camera for real-time prediction.

Default: `3600`

`ModelResourceKey`

(Optional) The models that are used in the dependent public model component. Modify this parameter only if you override the public model component with a custom component.

Default:

```
{
    "model": "TensorFlowLite-SSD"
}
```

`UseCamera`

(Optional) String value that defines whether to use images from a camera connected to the Greengrass core device. Supported values are `true` and `false`.

When you set this value to `true`, the sample inference code accesses the camera on your device and runs inference locally on the captured image. The values of the `ImageName` and `ImageDirectory` parameters are ignored. Make sure that the user running this component has read/write access to the location where the camera stores captured images.

Default: `false`

> **Note**
>
> When you view the recipe of this component, the `UseCamera` configuration parameter doesn't appear in the default configuration. However, you can modify the value of this parameter in a configuration merge update when you deploy the component.
> When you set `UseCamera` to `true`, you must also create a symlink to enable the inference component to access your camera from the virtual environment that is created by the runtime component. For more information about using a camera with the sample inference components, see Update component configurations.

Note

When you view the recipe of this component, the `UseCamera` configuration parameter doesn't appear in the default configuration. However, you can modify the value of this parameter in a configuration merge update when you deploy the component.
When you set `UseCamera` to `true`, you must also create a symlink to enable the inference component to access your camera from the virtual environment that is created by the runtime component. For more information about using a camera with the sample inference components, see Update component configurations.

Local log file

This component uses the following log file.

Linux

```
/greengrass/v2/logs/aws.greengrass.TensorFlowLiteObjectDetection.log
```

Windows

```
C:\greengrass\v2\logs\aws.greengrass.TensorFlowLiteObjectDetection.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/
  aws.greengrass.TensorFlowLiteObjectDetection.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs
  \aws.greengrass.TensorFlowLiteObjectDetection.log -Tail 10 -Wait
  ```

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.1.12	Version updated for Greengrass nucleus 2.12.5 release.
2.1.11	Version updated for Greengrass nucleus version 2.12.0 release.
2.1.10	Version updated for Greengrass nucleus version 2.11.0 release.
2.1.9	Version updated for Greengrass nucleus version 2.10.0 release.
2.1.8	Version updated for Greengrass nucleus version 2.9.0 release.
2.1.7	Version updated for Greengrass nucleus version 2.8.0 release.
2.1.6	Version updated for Greengrass nucleus version 2.7.0 release.
2.1.5	Version updated for Greengrass nucleus version 2.6.0 release.
2.1.4	Version updated for Greengrass nucleus version 2.5.0 release.
2.1.3	Version updated for Greengrass nucleus version 2.4.0 release.
2.1.2	Version updated for Greengrass nucleus version 2.3.0 release.
2.1.1	Bug fixes and improvements • Fixes an image scaling issue that resulted in inaccurate bounding boxes in the sample TensorFlow Lite object detection inference results.
2.1.0	Initial version.

TensorFlow Lite image classification model store

The TensorFlow Lite image classification model store (`variant.TensorFlowLite.ImageClassification.ModelStore`) is a machine learning model component that contains a pre-trained MobileNet v1 model as a Greengrass artifact. The

sample model used in this component is fetched from the TensorFlow Hub and implemented using TensorFlow Lite.

The TensorFlow Lite image classification inference component uses this component as a dependency for the model source. To use a custom-trained TensorFlow Lite model, create a custom version of this model component, and include your custom model as a component artifact. You can use the recipe of this component as a template to create custom model components.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Local log file
- Changelog

Versions

This component has the following versions:

- 2.1.x

Type

This component is a generic component (`aws.greengrass.generic`). The Greengrass nucleus runs the component's lifecycle scripts.

For more information, see Component types.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux

- Windows

Requirements

This component has the following requirements:

- On Greengrass core devices running Amazon Linux 2 or Ubuntu 18.04, GNU C Library (glibc) version 2.27 or later installed on the device.
- On Armv7l devices, such as Raspberry Pi, dependencies for OpenCV-Python installed on the device. Run the following command to install the dependencies.

  ```
  sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev
   libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
  ```
- Raspberry Pi devices that run Raspberry Pi OS Bullseye must meet the following requirements:
  - NumPy 1.22.4 or later installed on the device. Raspberry Pi OS Bullseye includes an earlier version of NumPy, so you can run the following command to upgrade NumPy on the device.

    ```
    pip3 install --upgrade numpy
    ```
  - The legacy camera stack enabled on the device. Raspberry Pi OS Bullseye includes a new camera stack that is enabled by default and isn't compatible, so you must enable the legacy camera stack.

    To enable the legacy camera stack

    1. Run the following command to open the Raspberry Pi configuration tool.

       ```
       sudo raspi-config
       ```
    2. Select **Interface Options**.
    3. Select **Legacy camera** to enable the legacy camera stack.
    4. Reboot the Raspberry Pi.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for

the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

2.1.11 and 2.1.12

The following table lists the dependencies for version 2.1.11 and 2.1.12 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.13.0	Soft

2.1.10

The following table lists the dependencies for version 2.1.10 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.12.0	Soft

2.1.9

The following table lists the dependencies for version 2.1.9 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.11.0	Soft

2.1.8

The following table lists the dependencies for version 2.1.8 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.10.0	Soft

2.1.7

The following table lists the dependencies for version 2.1.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.9.0	Soft

2.1.6

The following table lists the dependencies for version 2.1.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.8.0	Soft

2.1.5

The following table lists the dependencies for version 2.1.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.7.0	Soft

2.1.4

The following table lists the dependencies for version 2.1.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.6.0	Soft

2.1.3

The following table lists the dependencies for version 2.1.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.5.0	Soft

2.1.2

The following table lists the dependencies for version 2.1.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.4.0	Soft

2.1.1

The following table lists the dependencies for version 2.1.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.3.0	Soft

2.1.0

The following table lists the dependencies for version 2.1.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.2.0	Soft

Configuration

This component doesn't have any configuration parameters.

Local log file

This component doesn't output logs.

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.1.12	Version updated for Greengrass nucleus 2.12.5 release.
2.1.11	Version updated for Greengrass nucleus version 2.12.0 release.
2.1.10	Version updated for Greengrass nucleus version 2.11.0 release.
2.1.9	Version updated for Greengrass nucleus version 2.10.0 release.
2.1.8	Version updated for Greengrass nucleus version 2.9.0 release.
2.1.7	Version updated for Greengrass nucleus version 2.8.0 release.
2.1.6	Version updated for Greengrass nucleus version 2.7.0 release.
2.1.5	Version updated for Greengrass nucleus version 2.6.0 release.
2.1.4	Version updated for Greengrass nucleus version 2.5.0 release.
2.1.3	Version updated for Greengrass nucleus version 2.4.0 release.
2.1.2	Version updated for Greengrass nucleus version 2.3.0 release.
2.1.1	Version updated for Greengrass nucleus version 2.2.0 release.
2.1.0	Initial version.

TensorFlow Lite object detection model store

The TensorFlow Lite object detection model store (`variant.TensorFlowLite.ObjectDetection.ModelStore`) is a machine learning model component that contains a pre-trained Single Shot Detection (SSD) MobileNet model as a Greengrass artifact. The sample model used in this component is fetched from the TensorFlow Hub and implemented using TensorFlow Lite.

The TensorFlow Lite object detection inference component uses this component as a dependency for the model source. To use a custom-trained TensorFlow Lite model, create a custom version of this model component, and include your custom model as a component artifact. You can use the recipe of this component as a template to create custom model components.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Local log file
- Changelog

Versions

This component has the following versions:

- 2.1.x

Type

This component is a generic component (`aws.greengrass.generic`). The Greengrass nucleus runs the component's lifecycle scripts.

For more information, see Component types.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

Requirements

This component has the following requirements:

- On Greengrass core devices running Amazon Linux 2 or Ubuntu 18.04, GNU C Library (glibc) version 2.27 or later installed on the device.
- On Armv7l devices, such as Raspberry Pi, dependencies for OpenCV-Python installed on the device. Run the following command to install the dependencies.

  ```
  sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev
   libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
  ```
- Raspberry Pi devices that run Raspberry Pi OS Bullseye must meet the following requirements:
  - NumPy 1.22.4 or later installed on the device. Raspberry Pi OS Bullseye includes an earlier version of NumPy, so you can run the following command to upgrade NumPy on the device.

    ```
    pip3 install --upgrade numpy
    ```
  - The legacy camera stack enabled on the device. Raspberry Pi OS Bullseye includes a new camera stack that is enabled by default and isn't compatible, so you must enable the legacy camera stack.

    To enable the legacy camera stack

    1. Run the following command to open the Raspberry Pi configuration tool.

       ```
       sudo raspi-config
       ```
    2. Select **Interface Options**.
    3. Select **Legacy camera** to enable the legacy camera stack.
    4. Reboot the Raspberry Pi.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of

the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

2.1.11 and 2.1.12

The following table lists the dependencies for version 2.1.11 and 2.1.12 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.13.0	Soft

2.1.10

The following table lists the dependencies for version 2.1.10 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.12.0	Soft

2.1.9

The following table lists the dependencies for version 2.1.9 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.11.0	Soft

2.1.8

The following table lists the dependencies for version 2.1.8 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.10.0	Soft

2.1.7

The following table lists the dependencies for version 2.1.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.9.0	Soft

2.1.6

The following table lists the dependencies for version 2.1.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.8.0	Soft

2.1.5

The following table lists the dependencies for version 2.1.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.7.0	Soft

2.1.4

The following table lists the dependencies for version 2.1.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.6.0	Soft

2.1.3

The following table lists the dependencies for version 2.1.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.5.0	Soft

2.1.2

The following table lists the dependencies for version 2.1.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.4.0	Soft

2.1.1

The following table lists the dependencies for version 2.1.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.3.0	Soft

2.1.0

The following table lists the dependencies for version 2.1.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.2.0	Soft

Configuration

This component doesn't have any configuration parameters.

Local log file

This component doesn't output logs.

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.1.12	Version updated for Greengrass nucleus 2.12.5 release.
2.1.11	Version updated for Greengrass nucleus version 2.12.0 release.
2.1.10	Version updated for Greengrass nucleus version 2.11.0 release.
2.1.9	Version updated for Greengrass nucleus version 2.10.0 release.
2.1.8	Version updated for Greengrass nucleus version 2.9.0 release.
2.1.7	Version updated for Greengrass nucleus version 2.8.0 release.
2.1.6	Version updated for Greengrass nucleus version 2.7.0 release.
2.1.5	Version updated for Greengrass nucleus version 2.6.0 release.
2.1.4	Version updated for Greengrass nucleus version 2.5.0 release.
2.1.3	Version updated for Greengrass nucleus version 2.4.0 release.
2.1.2	Version updated for Greengrass nucleus version 2.3.0 release.
2.1.1	Version updated for Greengrass nucleus version 2.2.0 release.
2.1.0	Initial version.

TensorFlow Lite runtime

The TensorFlow Lite runtime component (`variant.TensorFlowLite`) contains a script that installs TensorFlow Lite version 2.5.0 and its dependencies in a virtual environment on your device. The TensorFlow Lite image classification and TensorFlow Lite object detection component use this runtime component as a dependency for installing TensorFlow Lite.

> **Note**
>
> TensorFlow Lite runtime component v2.5.6 and later reinstalls existing installations of the TensorFlow Lite runtime and its dependencies. This reinstallation helps to ensure that the core device runs compatible versions of TensorFlow Lite and its dependencies.

To use a different runtime, you can use the recipe of this component as a template to create a custom machine learning component.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Usage
- Local log file
- Changelog

Versions

This component has the following versions:

- 2.5.x

Type

This component is a generic component (`aws.greengrass.generic`). The Greengrass nucleus runs the component's lifecycle scripts.

For more information, see Component types.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

Requirements

This component has the following requirements:

- On Greengrass core devices running Amazon Linux 2 or Ubuntu 18.04, GNU C Library (glibc) version 2.27 or later installed on the device.
- On Armv7l devices, such as Raspberry Pi, dependencies for OpenCV-Python installed on the device. Run the following command to install the dependencies.

```
sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev
 libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
```

- Raspberry Pi devices that run Raspberry Pi OS Bullseye must meet the following requirements:
  - NumPy 1.22.4 or later installed on the device. Raspberry Pi OS Bullseye includes an earlier version of NumPy, so you can run the following command to upgrade NumPy on the device.

    ```
    pip3 install --upgrade numpy
    ```

  - The legacy camera stack enabled on the device. Raspberry Pi OS Bullseye includes a new camera stack that is enabled by default and isn't compatible, so you must enable the legacy camera stack.

    To enable the legacy camera stack

    1. Run the following command to open the Raspberry Pi configuration tool.

       ```
       sudo raspi-config
       ```

    2. Select **Interface Options**.
    3. Select **Legacy camera** to enable the legacy camera stack.
    4. Reboot the Raspberry Pi.

Endpoints and ports

By default, this component uses an installer script to install packages using the `apt`, `yum`, `brew`, and `pip` commands, depending on what platform the core device uses. This component must be able to perform outbound requests to various package indexes and repositories to run the installer script. To allow this component's outbound traffic through a proxy or firewall, you must identify the endpoints for the package indexes and repositories where your core device connects to install.

Consider the following when you identify endpoints required for this component's install script:

- The endpoints depend on the core device's platform. For example, a core device that runs Ubuntu uses `apt` rather than `yum` or `brew`. Additionally, devices that use the same package index might have different source lists, so they might retrieve packages from different repositories.
- The endpoints might differ between multiple devices that use the same package index, because each device has its own source lists that define where to retrieve packages.
- The endpoints might change over time. Each package index provides the URLs of the repositories where you download packages, and the owner of a package can change what URLs the package index provides.

For more information about the dependencies that this component installs, and how to disable the installer script, see the UseInstaller configuration parameter.

For more information about endpoints and ports required for basic operation, see Allow device traffic through a proxy or firewall.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

2.5.14 and 2.5.15

The following table lists the dependencies for version 2.5.14 and 2.5.15 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.13.0	Soft

2.5.13

The following table lists the dependencies for version 2.5.13 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.12.0	Soft

2.5.12

The following table lists the dependencies for version 2.5.12 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.11.0	Soft

2.5.11

The following table lists the dependencies for version 2.5.11 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.10.0	Soft

2.5.10

The following table lists the dependencies for version 2.5.10 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.9.0	Soft

2.5.9

The following table lists the dependencies for version 2.5.9 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.8.0	Soft

2.5.8

The following table lists the dependencies for version 2.5.8 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.7.0	Soft

2.5.5 - 2.5.7

The following table lists the dependencies for versions 2.5.5 through 2.5.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.6.0	Soft

2.5.3 and 2.5.4

The following table lists the dependencies for versions 2.5.3 and 2.5.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.5.0	Soft

2.5.2

The following table lists the dependencies for version 2.5.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.4.0	Soft

2.5.1

The following table lists the dependencies for version 2.5.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.3.0	Soft

2.5.0

The following table lists the dependencies for version 2.5.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.2.0	Soft

For more information about component dependencies, see the component recipe reference.

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

`MLRootPath`

(Optional) The path of the folder on Linux core devices where inference components read images and write inference results. You can modify this value to any location on your device to which the user running this component has read/write access.

Default: */greengrass/v2*`/work/variant.TensorFlowLite/greengrass_ml`

`WindowsMLRootPath`

This feature is available in v1.6.6 and later of this component.

(Optional) The path of the folder on Windows core device where inference components read images and write inference results. You can modify this value to any location on your device to which the user running this component has read/write access.

Default: *C:\greengrass\v2*\\work\\variant.DLR\\greengrass_ml

`UseInstaller`

(Optional) String value that defines whether to use the installer script in this component to install TensorFlow Lite and its dependencies. Supported values are `true` and `false`.

Set this value to `false` if you want to use a custom script for TensorFlow Lite installation, or if you want to include runtime dependencies in a pre-built Linux image. To use this component with the AWS-provided TensorFlow Lite inference components, install the following libraries, including any dependencies, and make them available to the system user, such as `ggc_user`, that runs the ML components.

- Python 3.8 or later, including `pip` for your version of Python
- TensorFlow Lite v2.5.0
- NumPy
- OpenCV-Python
- AWS IoT Device SDK v2 for Python
- AWS Common Runtime (CRT) Python
- Picamera (for Raspberry Pi devices)
- `awscam` module (for AWS DeepLens devices)
- libGL (for Linux devices)

Default: `true`

Usage

Use this component with the `UseInstaller` configuration parameter set to `true` to install TensorFlow Lite and its dependencies on your device. The component sets up a virtual environment on your device that includes the OpenCV and NumPy libraries that are required for TensorFlow Lite.

> **Note**
>
> The installer script in this component also installs the latest versions of additional system libraries that are required to configure the virtual environment on your device and to use

the installed machine learning framework. This might upgrade the existing system libraries on your device. Review the following table for the list of libraries that this component installs for each supported operating system. If you want to customize this installation process, set the `UseInstaller` configuration parameter to `false`, and develop your own installer script.

Platform	Libraries installed on the device system	Libraries installed in the virtual environment
Armv7l	`build-essential` , `cmake`, `ca-certificates` , `git`	`setuptools` , `wheel`
Amazon Linux 2	`mesa-libGL`	None
Ubuntu	`wget`	None

When you deploy your inference component, this runtime component first verifies if your device already has TensorFlow Lite and its dependencies installed. If not, then the runtime component installs them for you.

Local log file

This component uses the following log file.

Linux

```
/greengrass/v2/logs/variant.TensorFlowLite.log
```

Windows

```
C:\greengrass\v2\logs\variant.TensorFlowLite.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

Linux

```
sudo tail -f /greengrass/v2/logs/variant.TensorFlowLite.log
```

Windows (PowerShell)

```
Get-Content C:\greengrass\v2\logs\variant.TensorFlowLite.log -Tail 10 -Wait
```

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.5.15	Version updated for Greengrass nucleus 2.12.5 release.
2.5.14	Version updated for Greengrass nucleus version 2.12.0 release.
2.5.13	Version updated for Greengrass nucleus version 2.11.0 release.
2.5.12	Version updated for Greengrass nucleus version 2.10.0 release.
2.5.11	Version updated for Greengrass nucleus version 2.9.0 release.
2.5.10	Version updated for Greengrass nucleus version 2.8.0 release.
2.5.9	Version updated for Greengrass nucleus version 2.7.0 release.
2.5.8	Version updated for Greengrass nucleus version 2.6.0 release.
2.5.7	Bug fixes and improvements • Updates the `UseInstaller` installation script to install libGL, which isn't available by default on certain Linux platforms. • Updates the `UseInstaller` installation script to always use Python 3.9 in this component's virtual environment. This change helps ensure compatibility with other libraries.

Version	Changes
2.5.6	Bug fixes and improvements • Updates this component to install the latest patch of TensorFlow Lite 2.5.0 (`tflite-runtime-2.5.0.post1`), so you can use this component with Python 3.9. If this component fails to install that patch, it installs `tflite-runtime-2.5.0` instead. • Updates this component to reinstall existing installations of TensorFlow Lite and its dependencies. This change helps ensure that the core device runs compatible versions of TensorFlow Lite and its dependencies.
2.5.5	New features • Adds support for core devices that run Windows. • Adds the new `WindowsMLRootPath` configuration parameter that you can use to configure the inference results folder on Windows core devices.
2.5.4	New features • Adds the new `UseInstaller` configuration parameter that lets you disable the installation script in this component.
2.5.3	Version updated for Greengrass nucleus version 2.4.0 release.
2.5.2	Version updated for Greengrass nucleus version 2.3.0 release.
2.5.1	Version updated for Greengrass nucleus version 2.2.0 release.
2.5.0	Initial version.

Modbus-RTU protocol adapter

The Modbus-RTU protocol adapter component (`aws.greengrass.Modbus`) polls information from local Modbus RTU devices.

To request information from a local Modbus RTU device with this component, publish a message to the topic where this component subscribes. In the message, specify the Modbus RTU request to

send to a device. Then, this component publishes a response that contains the result of the Modbus RTU request.

> **Note**
>
> This component provides similar functionality to the Modbus RTU protocol adapter connector in AWS IoT Greengrass V1. For more information, see Modbus RTU protocol adapter connector in the *AWS IoT Greengrass V1 Developer Guide*.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Input data
- Output data
- Modbus RTU requests and responses
- Local log file
- Licenses
- Changelog

Versions

This component has the following versions:

- 2.1.x
- 2.0.x

Type

This component is a Lambda component (`aws.greengrass.lambda`). The Greengrass nucleus runs this component's Lambda function using the Lambda launcher component.

For more information, see Component types.

Operating system

This component can be installed on Linux core devices only.

Requirements

This component has the following requirements:

- Your core device must meet the requirements to run Lambda functions. If you want the core device to run containerized Lambda functions, the device must meet the requirements to do so. For more information, see Lambda function requirements.
- Python version 3.7 installed on the core device and added to the PATH environment variable.
- A physical connection between the AWS IoT Greengrass core device and the Modbus devices. The core device must be physically connected to the Modbus RTU network through a serial port, such as a USB port.
- To receive output data from this component, you must merge the following configuration update for the legacy subscription router component (`aws.greengrass.LegacySubscriptionRouter`) when you deploy this component. This configuration specifies the topic where this component publishes responses.

  Legacy subscription router v2.1.x

  ```
  {
    "subscriptions": {
      "aws-greengrass-modbus": {
        "id": "aws-greengrass-modbus",
        "source": "component:aws.greengrass.Modbus",
        "subject": "modbus/adapter/response",
        "target": "cloud"
      }
    }
  }
  ```

  Legacy subscription router v2.0.x

  ```
  {
    "subscriptions": {
      "aws-greengrass-modbus": {
        "id": "aws-greengrass-modbus",
  ```

```
      "source": "arn:aws:lambda:region:aws:function:aws-greengrass-
modbus:version",
      "subject": "modbus/adapter/response",
      "target": "cloud"
    }
  }
}
```

- Replace *region* with the AWS Region that you use.
- Replace *version* with the version of the Lambda function that this component runs. To find the Lambda function version, you must view the recipe for the version of this component that you want to deploy. Open this component's details page in the AWS IoT Greengrass console, and look for the **Lambda function** key-value pair. This key-value pair contains the name and version of the Lambda function.

> **Important**
>
> You must update the Lambda function version on the legacy subscription router every time you deploy this component. This ensures that you use the correct Lambda function version for the component version that you deploy.

For more information, see Create deployments.

- The Modbus-RTU protocol adapter is supported to run in a VPC.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

2.1.11

The following table lists the dependencies for version 2.1.11 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.16.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.1.10

The following table lists the dependencies for version 2.1.10 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.15.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.1.9

The following table lists the dependencies for version 2.1.9 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.14.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.1.8

The following table lists the dependencies for version 2.1.8 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.13.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.1.7

The following table lists the dependencies for version 2.1.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.12.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.1.6

The following table lists the dependencies for version 2.1.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.11.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft

Dependency	Compatible versions	Dependency type
Token exchange service	^2.0.0	Hard

2.1.4 and 2.1.5

The following table lists the dependencies for versions 2.1.4 and 2.1.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.10.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.1.3

The following table lists the dependencies for version 2.1.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.9.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.1.2

The following table lists the dependencies for version 2.1.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.8.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.1.1

The following table lists the dependencies for version 2.1.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.7.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.0.8 and 2.1.0

The following table lists the dependencies for versions 2.0.8 and 2.1.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.6.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.0.7

The following table lists the dependencies for version 2.0.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.5.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.0.6

The following table lists the dependencies for version 2.0.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.4.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.0.5

The following table lists the dependencies for version 2.0.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.3.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft

Dependency	Compatible versions	Dependency type
Token exchange service	^2.0.0	Hard

2.0.4

The following table lists the dependencies for version 2.0.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.2.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.0.3

The following table lists the dependencies for version 2.0.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.3 <2.1.0	Hard
Lambda launcher	>=1.0.0	Hard
Lambda runtimes	>=1.0.0	Soft
Token exchange service	>=1.0.0	Hard

For more information about component dependencies, see the component recipe reference.

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

> **Note**
>
> This component's default configuration includes Lambda function parameters. We recommend that you edit only the following parameters to configure this component on your devices.

v2.1.x

`lambdaParams`

An object that contains the parameters for this component's Lambda function. This object contains the following information:

`EnvironmentVariables`

An object that contains the Lambda function's parameters. This object contains the following information:

`ModbusLocalPort`

The absolute path to the physical Modbus serial port on the core device, such as `/dev/ttyS2`.

To run this component in a container, you must define this path as a system device (in `containerParams.devices`) that the component can access. This component runs in a container by default.

> **Note**
>
> This component must have read/write access to the device.

`ModbusBaudRate`

(Optional) A string value that specifies the baud rate for serial communication with local Modbus TCP devices.

Default: `9600`

`ModbusByteSize`

(Optional) A string value that specifies the size of a byte in serial communication with local Modbus TCP devices. Choose `5`, `6`, `7`, or `8` bits.

Default: `8`

`ModbusParity`

(Optional) The parity mode to use to verify data integrity in serial communication with local Modbus TCP devices.

- `E` – Verify data integrity with even parity.
- `0` – Verify data integrity with odd parity.
- `N` – Don't verify data integrity.

Default: `N`

`ModbusStopBits`

(Optional) A string value that specifies the number of bits that indicate the end of a byte in serial communication with local Modbus TCP devices.

Default: `1`

`containerMode`

(Optional) The containerization mode for this component. Choose from the following options:

- `GreengrassContainer` – The component runs in an isolated runtime environment inside the AWS IoT Greengrass container.

  If you specify this option, you must specify a system device (in `containerParams.devices`) to give the container access to the Modbus device.
- `NoContainer` – The component doesn't run in an isolated runtime environment.

Default: `GreengrassContainer`

`containerParams`

(Optional) An object that contains the container parameters for this component. The component uses these parameters if you specify `GreengrassContainer` for `containerMode`.

This object contains the following information:

`memorySize`

(Optional) The amount of memory (in kilobytes) to allocate to the component.

Defaults to 512 MB (525,312 KB).

`devices`

(Optional) An object that specifies the system devices that the component can access in a container.

> **Important**
>
> To run this component in a container, you must specify the system device that you configure in the `ModbusLocalPort` environment variable.

This object contains the following information:

`0` – This is an array index as a string.

An object that contains the following information:

`path`

The path to the system device on the core device. This must have the same value as the value that you configure for `ModbusLocalPort`.

`permission`

(Optional) The permission to access the system device from the container. This value must be `rw`, which specifies that the component has read/write access to the system device.

Default: `rw`

`addGroupOwner`

(Optional) Whether or not to add the system group that runs the component as an owner of the system device.

Default: `true`

`pubsubTopics`

(Optional) An object that contains the topics where the component subscribes to receive messages. You can specify each topic and whether the component subscribes to MQTT topics from AWS IoT Core or local publish/subscribe topics.

This object contains the following information:

`0` – This is an array index as a string.

An object that contains the following information:

`type`

(Optional) The type of publish/subscribe messaging that this component uses to subscribe to messages. Choose from the following options:

- PUB_SUB – Subscribe to local publish/subscribe messages. If you choose this option, the topic can't contain MQTT wildcards. For more information about how to send messages from custom component when you specify this option, see Publish/subscribe local messages.
- `IOT_CORE` – Subscribe to AWS IoT Core MQTT messages. If you choose this option, the topic can contain MQTT wildcards. For more information about how to send messages from custom components when you specify this option, see Publish/subscribe AWS IoT Core MQTT messages.

Default: PUB_SUB

`topic`

(Optional) The topic to which the component subscribes to receive messages. If you specify `IotCore` for `type`, you can use MQTT wildcards (+ and #) in this topic.

Example: Configuration merge update (container mode)

```
{
  "lambdaExecutionParameters": {
    "EnvironmentVariables": {
      "ModbusLocalPort": "/dev/ttyS2"
    }
  },
  "containerMode": "GreengrassContainer",
```

```
  "containerParams": {
    "devices": {
      "0": {
        "path": "/dev/ttyS2",
        "permission": "rw",
        "addGroupOwner": true
      }
    }
  }
}
```

Example: Configuration merge update (no container mode)

```
{
  "lambdaExecutionParameters": {
    "EnvironmentVariables": {
      "ModbusLocalPort": "/dev/ttyS2"
    }
  },
  "containerMode": "NoContainer"
}
```

v2.0.x

`lambdaParams`

An object that contains the parameters for this component's Lambda function. This object contains the following information:

`EnvironmentVariables`

An object that contains the Lambda function's parameters. This object contains the following information:

`ModbusLocalPort`

The absolute path to the physical Modbus serial port on the core device, such as `/dev/ttyS2`.

To run this component in a container, you must define this path as a system device (in `containerParams.devices`) that the component can access. This component runs in a container by default.

> **Note**
>
> This component must have read/write access to the device.

`containerMode`

(Optional) The containerization mode for this component. Choose from the following options:

- `GreengrassContainer` – The component runs in an isolated runtime environment inside the AWS IoT Greengrass container.

  If you specify this option, you must specify a system device (in `containerParams.devices`) to give the container access to the Modbus device.
- `NoContainer` – The component doesn't run in an isolated runtime environment.

Default: `GreengrassContainer`

`containerParams`

(Optional) An object that contains the container parameters for this component. The component uses these parameters if you specify `GreengrassContainer` for `containerMode`.

This object contains the following information:

`memorySize`

(Optional) The amount of memory (in kilobytes) to allocate to the component.

Defaults to 512 MB (525,312 KB).

`devices`

(Optional) An object that specifies the system devices that the component can access in a container.

> **Important**
>
> To run this component in a container, you must specify the system device that you configure in the `ModbusLocalPort` environment variable.

This object contains the following information:

`0` – This is an array index as a string.

An object that contains the following information:

`path`

The path to the system device on the core device. This must have the same value as the value that you configure for `ModbusLocalPort`.

`permission`

(Optional) The permission to access the system device from the container. This value must be `rw`, which specifies that the component has read/write access to the system device.

Default: `rw`

`addGroupOwner`

(Optional) Whether or not to add the system group that runs the component as an owner of the system device.

Default: `true`

`pubsubTopics`

(Optional) An object that contains the topics where the component subscribes to receive messages. You can specify each topic and whether the component subscribes to MQTT topics from AWS IoT Core or local publish/subscribe topics.

This object contains the following information:

`0` – This is an array index as a string.

An object that contains the following information:

`type`

(Optional) The type of publish/subscribe messaging that this component uses to subscribe to messages. Choose from the following options:

- PUB_SUB – Subscribe to local publish/subscribe messages. If you choose this option, the topic can't contain MQTT wildcards. For more information about how to send messages from custom component when you specify this option, see Publish/subscribe local messages.

- `IOT_CORE` – Subscribe to AWS IoT Core MQTT messages. If you choose this option, the topic can contain MQTT wildcards. For more information about how to send messages from custom components when you specify this option, see [Publish/subscribe AWS IoT Core MQTT messages].

  Default: PUB_SUB

`topic`

(Optional) The topic to which the component subscribes to receive messages. If you specify `IotCore` for `type`, you can use MQTT wildcards (+ and #) in this topic.

Example: Configuration merge update (container mode)

```
{
  "lambdaExecutionParameters": {
    "EnvironmentVariables": {
      "ModbusLocalPort": "/dev/ttyS2"
    }
  },
  "containerMode": "GreengrassContainer",
  "containerParams": {
    "devices": {
      "0": {
        "path": "/dev/ttyS2",
        "permission": "rw",
        "addGroupOwner": true
      }
    }
  }
}
```

Example: Configuration merge update (no container mode)

```
{
  "lambdaExecutionParameters": {
    "EnvironmentVariables": {
      "ModbusLocalPort": "/dev/ttyS2"
    }
  },
  "containerMode": "NoContainer"
}
```

Input data

This component accepts Modbus RTU request parameters on the following topic and sends the Modbus RTU request to the device. By default, this component subscribes to local publish/subscribe messages. For more information about how to publish messages to this component from your custom components, see Publish/subscribe local messages.

Default topic (local publish/subscribe): `modbus/adapter/request`

The message accepts the following properties. Input messages must be in JSON format.

`request`

The parameters for the Modbus RTU request to send.

The shape of the request message depends on the type of Modbus RTU request that it represents. The following properties are required for all requests.

Type: `object` that contains the following information:

`operation`

The name of the operation to run. For example, specify `ReadCoilsRequest` to read coils on a Modbus RTU device. For more information about supported operations, see Modbus RTU requests and responses.

Type: `string`

`device`

The target device of the request.

This value must be an integer between `0` and `247`.

Type: `integer`

The other parameters to include in the request depend on the operation. This component handles the cyclic redundancy check (CRC) to verify data requests for you.

> **Note**
>
> If you request includes an `address` property, you must specify its value as an integer. For example, `"address": 1`.

`id`

An arbitrary ID for the request. Use this property to map an input request to an output response. When you specify this property, the component sets the `id` property in the response object to this value.

Type: `string`

Example input: Read coils request

```
{
  "request": {
    "operation": "ReadCoilsRequest",
    "device": 1,
    "address": 1,
    "count": 1
  },
  "id": "MyRequest"
}
```

Output data

This component publishes responses as output data on the following MQTT topic by default. You must specify this topic as the `subject` in the configuration for the legacy subscription router component. For more information about how to subscribe to messages on this topic in your custom components, see Publish/subscribe AWS IoT Core MQTT messages.

Default topic (AWS IoT Core MQTT): `modbus/adapter/response`

The shape of the response message depends on the request operation and the response status. For examples, see Example requests and responses.

Every response includes the following properties:

`response`

The response from the Modbus RTU device.

Type: `object` that contains the following information:

`status`

The status of the request. The status can be one of the following values:

- `Success` – The request was valid, the component sent the request to the Modbus RTU network, and the Modbus RTU network returned a response.
- `Exception` – The request was valid, the component sent the request to the Modbus RTU network, and the Modbus RTU network returned an exception. For more information, see Response status: Exception.
- `No Response` – The request was invalid, and the component caught the error before it sent the request to the Modbus RTU network. For more information, see Response status: No response.

`operation`

The operation that the component requested.

`device`

The device where the component sent the request.

`payload`

The response from the Modbus RTU device. If the `status` is `No Response`, this object contains only an `error` property with the description of the error (for example, `[Input/Output] No Response received from the remote unit`).

`id`

The ID of the request, which you can use to identify which response corresponds to which request.

> **Note**
>
> A response for a write operation is simply an echo of the request. Although write responses don't include meaningful information, it's a good practice to check the status of the response to see if the request succeeds or fails.

Example output: Success

```
{
  "response" : {
    "status" : "success",
    "device": 1,
```

```
    "operation": "ReadCoilsRequest",
    "payload": {
      "function_code": 1,
      "bits": [1]
    }
  },
  "id" : "MyRequest"
}
```

Example output: Failure

```
{
  "response" : {
    "status" : "fail",
    "error_message": "Internal Error",
    "error": "Exception",
    "device": 1,
    "operation": "ReadCoilsRequest",
    "payload": {
      "function_code": 129,
      "exception_code": 2
    }
  },
  "id" : "MyRequest"
}
```

For more examples, see Example requests and responses.

Modbus RTU requests and responses

This connector accepts Modbus RTU request parameters as input data and publishes responses as output data.

The following common operations are supported.

Operation name in request	Function code in response
ReadCoilsRequest	01
ReadDiscreteInputsRequest	02
ReadHoldingRegistersRequest	03

Operation name in request	Function code in response
ReadInputRegistersRequest	04
WriteSingleCoilRequest	05
WriteSingleRegisterRequest	06
WriteMultipleCoilsRequest	15
WriteMultipleRegistersRequest	16
MaskWriteRegisterRequest	22
ReadWriteMultipleRegistersRequest	23

Example requests and responses

The following are example requests and responses for supported operations.

Read coils

Request example:

```
{
  "request": {
    "operation": "ReadCoilsRequest",
    "device": 1,
    "address": 1,
    "count": 1
  },
  "id": "TestRequest"
}
```

Response example:

```
{
  "response": {
    "status": "success",
    "device": 1,
    "operation": "ReadCoilsRequest",
```

```
    "payload": {
      "function_code": 1,
      "bits": [1]
    }
  },
  "id" : "TestRequest"
}
```

Read discrete inputs

Request example:

```
{
  "request": {
    "operation": "ReadDiscreteInputsRequest",
    "device": 1,
    "address": 1,
    "count": 1
  },
  "id": "TestRequest"
}
```

Response example:

```
{
  "response": {
    "status": "success",
    "device": 1,
    "operation": "ReadDiscreteInputsRequest",
    "payload": {
      "function_code": 2,
      "bits": [1]
    }
  },
  "id" : "TestRequest"
}
```

Read holding registers

Request example:

```
{
  "request": {
```

```
    "operation": "ReadHoldingRegistersRequest",
    "device": 1,
    "address": 1,
    "count": 1
  },
  "id": "TestRequest"
}
```

Response example:

```
{
  "response": {
    "status": "success",
    "device": 1,
    "operation": "ReadHoldingRegistersRequest",
    "payload": {
      "function_code": 3,
      "registers": [20,30]
    }
  },
  "id" : "TestRequest"
}
```

Read input registers

Request example:

```
{
  "request": {
    "operation": "ReadInputRegistersRequest",
    "device": 1,
    "address": 1,
    "count": 1
  },
  "id": "TestRequest"
}
```

Write single coil

Request example:

```
{
```

```
  "request": {
    "operation": "WriteSingleCoilRequest",
    "device": 1,
    "address": 1,
    "value": 1
  },
  "id": "TestRequest"
}
```

Response example:

```
{
  "response": {
    "status": "success",
    "device": 1,
    "operation": "WriteSingleCoilRequest",
    "payload": {
      "function_code": 5,
      "address": 1,
      "value": true
    }
  },
  "id" : "TestRequest"
}
```

Write single register

Request example:

```
{
  "request": {
    "operation": "WriteSingleRegisterRequest",
    "device": 1,
    "address": 1,
    "value": 1
  },
  "id": "TestRequest"
}
```

Write multiple coils

Request example:

```
{
  "request": {
    "operation": "WriteMultipleCoilsRequest",
    "device": 1,
    "address": 1,
    "values": [1,0,0,1]
  },
  "id": "TestRequest"
}
```

Response example:

```
{
  "response": {
    "status": "success",
    "device": 1,
    "operation": "WriteMultipleCoilsRequest",
    "payload": {
      "function_code": 15,
      "address": 1,
      "count": 4
    }
  },
  "id" : "TestRequest"
}
```

Write multiple registers

Request example:

```
{
  "request": {
    "operation": "WriteMultipleRegistersRequest",
    "device": 1,
    "address": 1,
    "values": [20,30,10]
  },
  "id": "TestRequest"
}
```

Response example:

```
{
  "response": {
    "status": "success",
    "device": 1,
    "operation": "WriteMultipleRegistersRequest",
    "payload": {
      "function_code": 23,
      "address": 1,
      "count": 3
    }
  },
  "id" : "TestRequest"
}
```

Mask write register

Request example:

```
{
  "request": {
    "operation": "MaskWriteRegisterRequest",
    "device": 1,
    "address": 1,
    "and_mask": 175,
    "or_mask": 1
  },
  "id": "TestRequest"
}
```

Response example:

```
{
  "response": {
    "status": "success",
    "device": 1,
    "operation": "MaskWriteRegisterRequest",
    "payload": {
      "function_code": 22,
      "and_mask": 0,
      "or_mask": 8
    }
  },
  "id" : "TestRequest"
```

```
}
```

Read write multiple registers

Request example:

```
{
  "request": {
    "operation": "ReadWriteMultipleRegistersRequest",
    "device": 1,
    "read_address": 1,
    "read_count": 2,
    "write_address": 3,
    "write_registers": [20,30,40]
  },
  "id": "TestRequest"
}
```

Response example:

```
{
  "response": {
    "status": "success",
    "device": 1,
    "operation": "ReadWriteMultipleRegistersRequest",
    "payload": {
      "function_code": 23,
      "registers": [10,20,10,20]
    }
  },
  "id" : "TestRequest"
}
```

> **Note**
>
> The response includes the registers that the component reads.

Response status: Exception

Exceptions can occur when the request format is valid, but the request is not completed successfully. In this case, the response contains the following information:

- The `status` is set to `Exception`.
- The `function_code` equals the function code of the request + 128.
- The `exception_code` contains the exception code. For more information, see Modbus exception codes.

Example:

```
{
  "response": {
    "status": "fail",
    "error_message": "Internal Error",
    "error": "Exception",
    "device": 1,
    "operation": "ReadCoilsRequest",
    "payload": {
      "function_code": 129,
      "exception_code": 2
    }
  },
  "id": "TestRequest"
}
```

Response status: No response

This connector performs validation checks on the Modbus request. For example, it checks for invalid formats and missing fields. If the validation fails, the connector doesn't send the request. Instead, it returns a response that contains the following information:

- The `status` is set to `No Response`.
- The `error` contains the error reason.
- The `error_message` contains the error message.

Examples:

```
{
  "response": {
    "status": "fail",
    "error_message": "Invalid address field. Expected <type 'int'>, got <type 'str'>",
```

```
    "error": "No Response",
    "device": 1,
    "operation": "ReadCoilsRequest",
    "payload": {
      "error": "Invalid address field. Expected Expected <type 'int'>, got <type
 'str'>"
    }
  },
  "id": "TestRequest"
}
```

If the request targets a nonexistent device or if the Modbus RTU network is not working, you might get a `ModbusIOException`, which uses the No Response format.

```
{
  "response": {
    "status": "fail",
    "error_message": "[Input/Output] No Response received from the remote unit",
    "error": "No Response",
    "device": 1,
    "operation": "ReadCoilsRequest",
    "payload": {
      "error": "[Input/Output] No Response received from the remote unit"
    }
  },
  "id": "TestRequest"
}
```

Local log file

This component uses the following log file.

```
/greengrass/v2/logs/aws.greengrass.Modbus.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* with the path to the AWS IoT Greengrass root folder.

  ```
  sudo tail -f /greengrass/v2/logs/aws.greengrass.Modbus.log
  ```

Licenses

This component includes the following third-party software/licensing:

- pymodbus/BSD License
- pyserial/BSD License

This component is released under the Greengrass Core Software License Agreement.

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.1.11	Version updated for Greengrass nucleus version 2.15.0 release.
2.1.10	Version updated for Greengrass nucleus version 2.14.0 release.
2.1.9	Version updated for Greengrass nucleus version 2.13.0 release.
2.1.8	Version updated for Greengrass nucleus version 2.12.0 release.
2.1.7	Version updated for Greengrass nucleus version 2.11.0 release.
2.1.6	Version updated for Greengrass nucleus version 2.10.0 release.
2.1.5	Bug fixes and improvements • Fixes an issue with the `ReadDiscreteInput` operation.
2.1.4	Version updated for Greengrass nucleus version 2.9.0 release.
2.1.3	Version updated for Greengrass nucleus version 2.8.0 release.
2.1.2	Version updated for Greengrass nucleus version 2.7.0 release.
2.1.1	Version updated for Greengrass nucleus version 2.6.0 release.

Version	Changes
2.1.0	New features • Adds the `ModbusBaudRate` , `ModbusByteSize` , `ModbusParity` , and `ModbusStopBits` options that you can specify to configure serial communication with Modbus RTU devices.
2.0.8	Version updated for Greengrass nucleus version 2.5.0 release.
2.0.7	Version updated for Greengrass nucleus version 2.4.0 release.
2.0.6	Version updated for Greengrass nucleus version 2.3.0 release.
2.0.5	Version updated for Greengrass nucleus version 2.2.0 release.
2.0.4	Version updated for Greengrass nucleus version 2.1.0 release.
2.0.3	Initial version.

MQTT bridge

The MQTT bridge component (`aws.greengrass.clientdevices.mqtt.Bridge`) relays MQTT messages between client devices, local Greengrass publish/subscribe, and AWS IoT Core. You can use this component to act on MQTT messages from client devices in custom components and sync client devices with the AWS Cloud.

> **Note**
>
> Client devices are local IoT devices that connect to a Greengrass core device to send MQTT messages and data to process. For more information, see Interact with local IoT devices.

You can use this component to relay messages between the following message brokers:

- Local MQTT – The local MQTT broker handles messages between client devices and a core device.
- Local publish/subscribe – The local Greengrass message broker handles messages between components on a core device. For more information about how to interact with these messages in Greengrass components, see Publish/subscribe local messages.

- AWS IoT Core – The AWS IoT Core MQTT broker handles messages between IoT devices and AWS Cloud destinations. For more information about how to interact with these messages in Greengrass components, see Publish/subscribe AWS IoT Core MQTT messages.

> **Note**
>
> The MQTT bridge uses QoS 1 to publish and subscribe to AWS IoT Core, even when a client device uses QoS 0 to publish and subscribe to the local MQTT broker. As a result, you might observe additional latency when you relay MQTT messages from client devices on the local MQTT broker to AWS IoT Core. For more information about MQTT configuration on core devices, see Configure MQTT timeouts and cache settings.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Local log file
- Changelog

Versions

This component has the following versions:

- 2.3.x
- 2.2.x
- 2.1.x
- 2.0.x

Type

This component is a plugin component (`aws.greengrass.plugin`). The Greengrass nucleus runs this component in the same Java Virtual Machine (JVM) as the nucleus. The nucleus restarts when you change this component's version on the core device.

This component uses the same log file as the Greengrass nucleus. For more information, see Monitor AWS IoT Greengrass logs.

For more information, see Component types.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

Requirements

This component has the following requirements:

- If you configure the core device's MQTT broker component to use a port other than the default port 8883, you must use MQTT bridge v2.1.0 or later. Configure it to connect on the port where the broker operates.
- The MQTT bridge component is supported to run in a VPC.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

2.3.2

The following table lists the dependencies for version 2.3.2 of this component.

Dependency	Compatible versions	Dependency type
Client device auth	>=2.2.0 <2.6.0	Hard

2.3.0 and 2.3.1

The following table lists the dependencies for version 2.3.0 and 2.3.1 of this component.

Dependency	Compatible versions	Dependency type
Client device auth	>=2.2.0 <2.5.0	Hard

2.2.5 and 2.2.6

The following table lists the dependencies for version 2.2.5 and 2.2.6 of this component.

Dependency	Compatible versions	Dependency type
Client device auth	>=2.2.0 <2.5.0	Hard

2.2.3 and 2.2.4

The following table lists the dependencies for versions 2.2.3 and 2.2.4 of this component.

Dependency	Compatible versions	Dependency type
Client device auth	>=2.2.0 <2.4.0	Hard

2.2.0 – 2.2.2

The following table lists the dependencies for versions 2.2.0 to 2.2.2 of this component.

Dependency	Compatible versions	Dependency type
Client device auth	>=2.2.0 <2.3.0	Hard

2.1.1

The following table lists the dependencies for version 2.1.1 of this component.

Dependency	Compatible versions	Dependency type
Client device auth	>=2.0.0 <2.2.0	Hard

2.0.0 to 2.1.0

The following table lists the dependencies for versions 2.0.0 through 2.1.0 of this component.

Dependency	Compatible versions	Dependency type
Client device auth	>=2.0.0 <2.1.0	Hard

For more information about component dependencies, see the component recipe reference.

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

2.3.0 – 2.3.2

`mqttTopicMapping`

The topic mappings that you want to bridge. This component subscribes to messages on the source topic and publishes the messages that it receives to the destination topic. Each topic mapping defines the topic, source type, and destination type.

This object contains the following information:

topicMappingNameKey

The name of this topic mapping. Replace *topicMappingNameKey* with a name that helps you identify this topic mapping.

This object contains the following information:

`topic`

The topic or topic filter to bridge between the source and target brokers.

You can use the + and # MQTT topic wildcards to relay messages on all topics that match a topic filter. For more information, see MQTT topics in the *AWS IoT Core Developer Guide*.

> **Note**
>
> To use MQTT topic wildcards with the `Pubsub` source broker, you must use v2.6.0 or later of the Greengrass nucleus component.

`targetTopicPrefix`

The prefix to add to the target topic when this component relays the message.

`source`

The source message broker. Choose from the following options:

- `LocalMqtt` – The local MQTT broker where client devices communicate.
- `Pubsub` – The local Greengrass publish/subscribe message broker.
- `IotCore` – The AWS IoT Core MQTT message broker.

> **Note**
>
> The MQTT bridge uses QoS 1 to publish and subscribe to AWS IoT Core, even when a client device uses QoS 0 to publish and subscribe to the local MQTT broker. As a result, you might observe additional latency when you relay MQTT messages from client devices on the local MQTT broker to AWS IoT Core. For more information about MQTT configuration on core devices, see Configure MQTT timeouts and cache settings.

`source` and `target` must be different.

`target`

The target message broker. Choose from the following options:

- `LocalMqtt` – The local MQTT broker where client devices communicate.
- `Pubsub` – The local Greengrass publish/subscribe message broker.
- `IotCore` – The AWS IoT Core MQTT message broker.

> **Note**
>
> The MQTT bridge uses QoS 1 to publish and subscribe to AWS IoT Core, even when a client device uses QoS 0 to publish and subscribe to the local MQTT broker. As a result, you might observe additional latency when you relay MQTT messages from client devices on the local MQTT broker to AWS IoT Core. For more information about MQTT configuration on core devices, see Configure MQTT timeouts and cache settings.

`source` and `target` must be different.

mqtt5RouteOptions

(Optional) Provides options for configuring topic mappings for bridging messages from the source topic to the destination topic.

This object contains the following information:

`mqtt5RouteOptionsNameKey`

The name of the route options for a topic mapping. Replace *`mqtt5RouteOptionsNameKey`* with the matching *`topicMappingNameKey`* defined in the `mqttTopicMapping` field.

This object contains the following information:

noLocal

(Optional) When enabled, the bridge doesn't forward messages on a topic that the bridge itself published. Use this to prevent loops, as follows:

```
{
```

```
    "mqtt5RouteOptions": {
        "toIoTCore": {
            "noLocal": true
        }
    },
    "mqttTopicMapping": {
        "toIoTCore": {
            "topic": "device",
            "source": "LocalMqtt",
            "target": "IotCore"
        },
        "toLocal": {
            "topic": "device",
            "source": "IotCore",
            "target": "LocalMqtt"
        }
    }
}
```

noLocal is only supported for routes where the source is LocalMqtt.

Default: false

retainAsPublished

(Optional) When enabled, messages forwarded by the bridge have the same retain flag as messages published to the broker for that route.

retainAsPublished is only supported for routes where the source is LocalMqtt.

Default: false

mqtt

(Optional) MQTT protocol settings for communicating with the local broker.

version

(Optional) The MQTT protocol version used by the bridge to communicate with the local broker. Must be the same as the MQTT version selected in the nucleus configuration.

Choose from the following:

- mqtt3
- mqtt5

You must deploy an MQTT broker when the `source` or `target` field of the `mqttTopicMapping` object is set to `LocalMqtt`. If you choose the `mqtt5` option you must use the MQTT 5 broker (EMQX).

Default: `mqtt3`

ackTimeoutSeconds

(Optional) Time interval to wait for PUBACK, SUBACK, or UNSUBACK packets before failing the operation.

Default: 60

connAckTimeoutMs

(Optional) Time interval to wait for a CONNACK packet before shutting down the connection.

Default: 20000 (20 seconds)

pingTimeoutMs

(Optional) The amount of time in milliseconds that the bridge waits to receive a PINGACK message from the local broker. If the wait exceeds the timeout, the bridge closes then reopens the MQTT connection. This value must be less than `keepAliveTimeoutSeconds`.

Default: 30000 (30 seconds)

keepAliveTimeoutSeconds

(Optional) The amount of time in seconds between each PING message that the bridge sends to keep the MQTT connection alive. This value must be greater than `pingTimeoutMs`.

Default: 60

maxReconnectDelayMs

(Optional) The maximum amount of time in seconds for MQTT to reconnect.

Default: 30000 (30 seconds)

minReconnectDelayMs

(Optional) The minimum amount of time in seconds for MQTT to reconnect.

receiveMaximum

(Optional) The maximum number of unacknowledged QoS1 packets the bridge can send.

Default: 100

maximumPacketSize

The maximum number of bytes the client will accept for an MQTT packet.

Default: null (No limit)

sessionExpiryInterval

(Optional) The amount of time in seconds you can request for a session to last between the bridge and the local broker.

Default: 4294967295 (session never expires)

`brokerUri`

(Optional) The URI of the local MQTT broker. You must specify this parameter if you configure the MQTT broker to use a different port than the default port 8883. Use the following format, and replace *`port`* with the port where the MQTT broker operates: `ssl://localhost:`*`port`*.

Default: `ssl://localhost:8883`

startupTimeoutSeconds

(Optional) The maximum of time in seconds for the component to start. The component's state changes to `BROKEN` if it exceeds this timeout.

Default: `120`

Example: Configuration merge update

The following example configuration update specifies the following:

- Relay messages from client devices to AWS IoT Core on topics that match the `clients/+/hello/world` topic filter.
- Relay messages from client devices to local publish/subscribe on topics that match the `clients/+/detections` topic filter, and add the `events/input/` prefix to the target

topic. The resulting target topic matches the `events/input/clients/+/detections` topic filter.

- Relay messages from client devices to AWS IoT Core on topics that match the `clients/+/status` topic filter, and add the `$aws/rules/StatusUpdateRule/` prefix to the target topic. This example relays these messages directly to an AWS IoT rule named `StatusUpdateRule` to reduce costs using Basic Ingest.

```
{
  "mqttTopicMapping": {
    "ClientDeviceHelloWorld": {
      "topic": "clients/+/hello/world",
      "source": "LocalMqtt",
      "target": "IotCore"
    },
    "ClientDeviceEvents": {
      "topic": "clients/+/detections",
      "targetTopicPrefix": "events/input/",
      "source": "LocalMqtt",
      "target": "Pubsub"
    },
    "ClientDeviceCloudStatusUpdate": {
      "topic": "clients/+/status",
      "targetTopicPrefix": "$aws/rules/StatusUpdateRule/",
      "source": "LocalMqtt",
      "target": "IotCore"
    }
  }
}
```

Example: Configuring MQTT 5

The following example configuration updates the following:

- Enables the bridge to use the MQTT 5 protocol with the local broker.
- Configures MQTT retain as published setting for the `ClientDeviceHelloWorld` topic mapping.

```
{
  "mqttTopicMapping": {
```

```
    "ClientDeviceHelloWorld": {
      "topic": "clients/+/hello/world",
      "source": "LocalMqtt",
      "target": "IotCore"
    }
  },
  "mqtt5RouteOptions": {
    "ClientDeviceHelloWorld": {
      "retainAsPublished": true
    }
  },
  "mqtt": {
    "version": "mqtt5"
  }
}
```

2.2.6

`mqttTopicMapping`

The topic mappings that you want to bridge. This component subscribes to messages on the source topic and publishes the messages that it receives to the destination topic. Each topic mapping defines the topic, source type, and destination type.

This object contains the following information:

`topicMappingNameKey`

The name of this topic mapping. Replace *`topicMappingNameKey`* with a name that helps you identify this topic mapping.

This object contains the following information:

`topic`

The topic or topic filter to bridge between the source and target brokers.

You can use the + and # MQTT topic wildcards to relay messages on all topics that match a topic filter. For more information, see MQTT topics in the *AWS IoT Core Developer Guide*.

> **Note**
>
> To use MQTT topic wildcards with the `Pubsub` source broker, you must use v2.6.0 or later of the Greengrass nucleus component.

`targetTopicPrefix`

The prefix to add to the target topic when this component relays the message.

`source`

The source message broker. Choose from the following options:

- `LocalMqtt` – The local MQTT broker where client devices communicate.
- `Pubsub` – The local Greengrass publish/subscribe message broker.
- `IotCore` – The AWS IoT Core MQTT message broker.

> **Note**
>
> The MQTT bridge uses QoS 1 to publish and subscribe to AWS IoT Core, even when a client device uses QoS 0 to publish and subscribe to the local MQTT broker. As a result, you might observe additional latency when you relay MQTT messages from client devices on the local MQTT broker to AWS IoT Core. For more information about MQTT configuration on core devices, see Configure MQTT timeouts and cache settings.

`source` and `target` must be different.

`target`

The target message broker. Choose from the following options:

- `LocalMqtt` – The local MQTT broker where client devices communicate.
- `Pubsub` – The local Greengrass publish/subscribe message broker.
- `IotCore` – The AWS IoT Core MQTT message broker.

> **Note**
>
> The MQTT bridge uses QoS 1 to publish and subscribe to AWS IoT Core, even when a client device uses QoS 0 to publish and subscribe to the local

MQTT broker. As a result, you might observe additional latency when you relay MQTT messages from client devices on the local MQTT broker to AWS IoT Core. For more information about MQTT configuration on core devices, see Configure MQTT timeouts and cache settings.

`source` and `target` must be different.

`brokerUri`

(Optional) The URI of the local MQTT broker. You must specify this parameter if you configure the MQTT broker to use a different port than the default port 8883. Use the following format, and replace *`port`* with the port where the MQTT broker operates: `ssl://localhost:`*`port`*.

Default: `ssl://localhost:8883`

startupTimeoutSeconds

(Optional) The maximum of time in seconds for the component to start. The component's state changes to BROKEN if it exceeds this timeout.

Default: `120`

Example: Configuration merge update

The following example configuration update specifies the following:

- Relay messages from client devices to AWS IoT Core on topics that match the `clients/+/hello/world` topic filter.
- Relay messages from client devices to local publish/subscribe on topics that match the `clients/+/detections` topic filter, and add the `events/input/` prefix to the target topic. The resulting target topic matches the `events/input/clients/+/detections` topic filter.
- Relay messages from client devices to AWS IoT Core on topics that match the `clients/+/status` topic filter, and add the `$aws/rules/StatusUpdateRule/` prefix to the target topic. This example relays these messages directly to an AWS IoT rule named `StatusUpdateRule` to reduce costs using Basic Ingest.

```
{
```

```
  "mqttTopicMapping": {
    "ClientDeviceHelloWorld": {
      "topic": "clients/+/hello/world",
      "source": "LocalMqtt",
      "target": "IotCore"
    },
    "ClientDeviceEvents": {
      "topic": "clients/+/detections",
      "targetTopicPrefix": "events/input/",
      "source": "LocalMqtt",
      "target": "Pubsub"
    },
    "ClientDeviceCloudStatusUpdate": {
      "topic": "clients/+/status",
      "targetTopicPrefix": "$aws/rules/StatusUpdateRule/",
      "source": "LocalMqtt",
      "target": "IotCore"
    }
  }
}
```

2.2.0 - 2.2.5

`mqttTopicMapping`

The topic mappings that you want to bridge. This component subscribes to messages on the source topic and publishes the messages that it receives to the destination topic. Each topic mapping defines the topic, source type, and destination type.

This object contains the following information:

`topicMappingNameKey`

The name of this topic mapping. Replace *`topicMappingNameKey`* with a name that helps you identify this topic mapping.

This object contains the following information:

`topic`

The topic or topic filter to bridge between the source and target brokers.

You can use the + and # MQTT topic wildcards to relay messages on all topics that match a topic filter. For more information, see MQTT topics in the *AWS IoT Core Developer Guide*.

> **Note**
>
> To use MQTT topic wildcards with the `Pubsub` source broker, you must use v2.6.0 or later of the Greengrass nucleus component.

`targetTopicPrefix`

The prefix to add to the target topic when this component relays the message.

`source`

The source message broker. Choose from the following options:

- `LocalMqtt` – The local MQTT broker where client devices communicate.
- `Pubsub` – The local Greengrass publish/subscribe message broker.
- `IotCore` – The AWS IoT Core MQTT message broker.

> **Note**
>
> The MQTT bridge uses QoS 1 to publish and subscribe to AWS IoT Core, even when a client device uses QoS 0 to publish and subscribe to the local MQTT broker. As a result, you might observe additional latency when you relay MQTT messages from client devices on the local MQTT broker to AWS IoT Core. For more information about MQTT configuration on core devices, see Configure MQTT timeouts and cache settings.

`source` and `target` must be different.

`target`

The target message broker. Choose from the following options:

- `LocalMqtt` – The local MQTT broker where client devices communicate.
- `Pubsub` – The local Greengrass publish/subscribe message broker.
- `IotCore` – The AWS IoT Core MQTT message broker.

> **Note**
>
> The MQTT bridge uses QoS 1 to publish and subscribe to AWS IoT Core, even when a client device uses QoS 0 to publish and subscribe to the local

MQTT broker. As a result, you might observe additional latency when you relay MQTT messages from client devices on the local MQTT broker to AWS IoT Core. For more information about MQTT configuration on core devices, see Configure MQTT timeouts and cache settings.

`source` and `target` must be different.

`brokerUri`

(Optional) The URI of the local MQTT broker. You must specify this parameter if you configure the MQTT broker to use a different port than the default port 8883. Use the following format, and replace *`port`* with the port where the MQTT broker operates: `ssl://localhost:`*`port`*.

Default: `ssl://localhost:8883`

Example: Configuration merge update

The following example configuration update specifies the following:

- Relay messages from client devices to AWS IoT Core on topics that match the `clients/+/hello/world` topic filter.
- Relay messages from client devices to local publish/subscribe on topics that match the `clients/+/detections` topic filter, and add the `events/input/` prefix to the target topic. The resulting target topic matches the `events/input/clients/+/detections` topic filter.
- Relay messages from client devices to AWS IoT Core on topics that match the `clients/+/status` topic filter, and add the `$aws/rules/StatusUpdateRule/` prefix to the target topic. This example relays these messages directly to an AWS IoT rule named `StatusUpdateRule` to reduce costs using Basic Ingest.

```
{
  "mqttTopicMapping": {
    "ClientDeviceHelloWorld": {
      "topic": "clients/+/hello/world",
      "source": "LocalMqtt",
      "target": "IotCore"
    },
```

```
    "ClientDeviceEvents": {
      "topic": "clients/+/detections",
      "targetTopicPrefix": "events/input/",
      "source": "LocalMqtt",
      "target": "Pubsub"
    },
    "ClientDeviceCloudStatusUpdate": {
      "topic": "clients/+/status",
      "targetTopicPrefix": "$aws/rules/StatusUpdateRule/",
      "source": "LocalMqtt",
      "target": "IotCore"
    }
  }
}
```

2.1.x

`mqttTopicMapping`

The topic mappings that you want to bridge. This component subscribes to messages on the source topic and publishes the messages that it receives to the destination topic. Each topic mapping defines the topic, source type, and destination type.

This object contains the following information:

`topicMappingNameKey`

The name of this topic mapping. Replace *`topicMappingNameKey`* with a name that helps you identify this topic mapping.

This object contains the following information:

`topic`

The topic or topic filter to bridge between the source and target brokers.

If you specify the `LocalMqtt` or `IotCore` source broker, you can use the + and # MQTT topic wildcards to relay messages on all topics that match a topic filter. For more information, see MQTT topics in the *AWS IoT Core Developer Guide*.

`source`

The source message broker. Choose from the following options:

- `LocalMqtt` – The local MQTT broker where client devices communicate.
- `Pubsub` – The local Greengrass publish/subscribe message broker.
- `IotCore` – The AWS IoT Core MQTT message broker.

> **Note**
>
> The MQTT bridge uses QoS 1 to publish and subscribe to AWS IoT Core, even when a client device uses QoS 0 to publish and subscribe to the local MQTT broker. As a result, you might observe additional latency when you relay MQTT messages from client devices on the local MQTT broker to AWS IoT Core. For more information about MQTT configuration on core devices, see Configure MQTT timeouts and cache settings.

`source` and `target` must be different.

`target`

The target message broker. Choose from the following options:

- `LocalMqtt` – The local MQTT broker where client devices communicate.
- `Pubsub` – The local Greengrass publish/subscribe message broker.
- `IotCore` – The AWS IoT Core MQTT message broker.

> **Note**
>
> The MQTT bridge uses QoS 1 to publish and subscribe to AWS IoT Core, even when a client device uses QoS 0 to publish and subscribe to the local MQTT broker. As a result, you might observe additional latency when you relay MQTT messages from client devices on the local MQTT broker to AWS IoT Core. For more information about MQTT configuration on core devices, see Configure MQTT timeouts and cache settings.

`source` and `target` must be different.

`brokerUri`

(Optional) The URI of the local MQTT broker. You must specify this parameter if you configure the MQTT broker to use a different port than the default port 8883. Use the

following format, and replace *port* with the port where the MQTT broker operates: `ssl://localhost:`*port*.

Default: `ssl://localhost:8883`

Example: Configuration merge update

The following example configuration update specifies to relay messages from client devices to AWS IoT Core on the `clients/MyClientDevice1/hello/world` and `clients/MyClientDevice2/hello/world` topics.

```
{
  "mqttTopicMapping": {
    "ClientDevice1HelloWorld": {
      "topic": "clients/MyClientDevice1/hello/world",
      "source": "LocalMqtt",
      "target": "IotCore"
    },
    "ClientDevice2HelloWorld": {
      "topic": "clients/MyClientDevice2/hello/world",
      "source": "LocalMqtt",
      "target": "IotCore"
    }
  }
}
```

2.0.x

`mqttTopicMapping`

The topic mappings that you want to bridge. This component subscribes to messages on the source topic and publishes the messages that it receives to the destination topic. Each topic mapping defines the topic, source type, and destination type.

This object contains the following information:

`topicMappingNameKey`

The name of this topic mapping. Replace *`topicMappingNameKey`* with a name that helps you identify this topic mapping.

This object contains the following information:

`topic`

The topic or topic filter to bridge between the source and target brokers.

If you specify the `LocalMqtt` or `IotCore` source broker, you can use the + and # MQTT topic wildcards to relay messages on all topics that match a topic filter. For more information, see MQTT topics in the *AWS IoT Core Developer Guide*.

`source`

The source message broker. Choose from the following options:

- `LocalMqtt` – The local MQTT broker where client devices communicate.
- `Pubsub` – The local Greengrass publish/subscribe message broker.
- `IotCore` – The AWS IoT Core MQTT message broker.

> **Note**
>
> The MQTT bridge uses QoS 1 to publish and subscribe to AWS IoT Core, even when a client device uses QoS 0 to publish and subscribe to the local MQTT broker. As a result, you might observe additional latency when you relay MQTT messages from client devices on the local MQTT broker to AWS IoT Core. For more information about MQTT configuration on core devices, see Configure MQTT timeouts and cache settings.

`source` and `target` must be different.

`target`

The target message broker. Choose from the following options:

- `LocalMqtt` – The local MQTT broker where client devices communicate.
- `Pubsub` – The local Greengrass publish/subscribe message broker.
- `IotCore` – The AWS IoT Core MQTT message broker.

> **Note**
>
> The MQTT bridge uses QoS 1 to publish and subscribe to AWS IoT Core, even when a client device uses QoS 0 to publish and subscribe to the local MQTT broker. As a result, you might observe additional latency when you

relay MQTT messages from client devices on the local MQTT broker to AWS IoT Core. For more information about MQTT configuration on core devices, see Configure MQTT timeouts and cache settings.

`source` and `target` must be different.

Example: Configuration merge update

The following example configuration update specifies to relay messages from client devices to AWS IoT Core on the `clients/MyClientDevice1/hello/world` and `clients/MyClientDevice2/hello/world` topics.

```
{
  "mqttTopicMapping": {
    "ClientDevice1HelloWorld": {
      "topic": "clients/MyClientDevice1/hello/world",
      "source": "LocalMqtt",
      "target": "IotCore"
    },
    "ClientDevice2HelloWorld": {
      "topic": "clients/MyClientDevice2/hello/world",
      "source": "LocalMqtt",
      "target": "IotCore"
    }
  }
}
```

Local log file

This component uses the same log file as the Greengrass nucleus component.

Linux

```
/greengrass/v2/logs/greengrass.log
```

Windows

```
C:\greengrass\v2\logs\greengrass.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
  ```

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.3.3	Bug fixes and improvements • Fixes an issue where high concurrent device load could cause the MQTT bridge to stop relaying messages after a dropped PUBACK.
2.3.2	Version updated for client device auth version 2.5.0 release.
2.3.1	Bug fixes and improvements Fixes an issue where the local MQTT client gets into a disconnect loop.
2.3.0	New features Adds MQTT5 support for bridging between AWS IoT Core and local MQTT sources.
2.2.6	New features Adds a new `startupTimeoutSeconds` configuration option.

Version	Changes
2.2.5	Version updated for client device auth version 2.4.0 release.
2.2.4	Version updated for Greengrass client device auth version 2.3.0 release.
2.2.3	This version contains bug fixes and improvements.
2.2.2	Bug fixes and improvements • Logging adjustments.
2.2.1	Bug fixes and improvements Fixes issues that can result in the MQTT bridge failing to subscribe to MQTT topics.
2.2.0	New features • Adds support for MQTT topic wildcards (# and +) when you specify local publish/subscribe as the source message broker. This feature requires v2.6.0 or later of the Greengrass nucleus component. • Adds the `targetTopicPrefix` option, which you can specify to configure the MQTT bridge to add a prefix to the target topic when it relays a message.
2.1.1	Bug fixes and improvements • Fixes issues with how this component handles configuration reset updates. • Reduces the frequency of MQTT client disconnects when certificates rotate.
2.1.0	New features • Adds the `brokerUri` parameter, which enables you to use a non-default MQTT broker port.
2.0.1	This version includes bug fixes and improvements.

Version	Changes
2.0.0	Initial version.

MQTT 3.1.1 broker (Moquette)

The Moquette MQTT broker component (`aws.greengrass.clientdevices.mqtt.Moquette`) handles MQTT messages between client devices and a Greengrass core device. This component provides a modified version of the Moquette MQTT broker. Deploy this MQTT broker to run a lightweight MQTT broker. For more information about how to choose an MQTT broker, see Choose an MQTT broker.

This broker implements the MQTT 3.1.1 protocol. It includes support for QoS 0, QoS 1, QoS 2 retained messages, last will messages, and persistent sessions.

> **Note**
>
> Client devices are local IoT devices that connect to a Greengrass core device to send MQTT messages and data to process. For more information, see Interact with local IoT devices.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Local log file
- Changelog

Versions

This component has the following versions:

- 2.3.x

- 2.2.x
- 2.1.x
- 2.0.x

Type

This component is a plugin component (`aws.greengrass.plugin`). The Greengrass nucleus runs this component in the same Java Virtual Machine (JVM) as the nucleus. The nucleus restarts when you change this component's version on the core device.

This component uses the same log file as the Greengrass nucleus. For more information, see Monitor AWS IoT Greengrass logs.

For more information, see Component types.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

Requirements

This component has the following requirements:

- The core device must be able to accept connections on the port where the MQTT broker operates. This component runs the MQTT broker on port 8883 by default. You can specify a different port when you configure this component.

  If you specify a different port, and you use the MQTT bridge component to relay MQTT messages to other brokers, you must use MQTT bridge v2.1.0 or later. Configure it to use the port where the MQTT broker operates.

  If you specify a different port, and you use the IP detector component to manage MQTT broker endpoints, you must use IP detector v2.1.0 or later. Configure it to report the port where the MQTT broker operates.
- The Moquette MQTT broker component is supported to run in a VPC.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

2.3.7

The following table lists the dependencies for version 2.3.7 of this component.

Dependency	Compatible versions	Dependency type
Client device auth	>=2.2.0 <2.6.0	Hard

2.3.2 – 2.3.6

The following table lists the dependencies for versions 2.3.2 through 2.3.6 of this component.

Dependency	Compatible versions	Dependency type
Client device auth	>=2.2.0 <2.5.0	Hard

2.3.0 and 2.3.1

The following table lists the dependencies for versions 2.3.0 and 2.3.1 of this component.

Dependency	Compatible versions	Dependency type
Client device auth	>=2.2.0 <2.4.0	Hard

2.2.0

The following table lists the dependencies for version 2.2.0 of this component.

Dependency	Compatible versions	Dependency type
Client device auth	>=2.2.0 <2.3.0	Hard

2.1.0

The following table lists the dependencies for version 2.1.0 of this component.

Dependency	Compatible versions	Dependency type
Client device auth	>=2.0.0 <2.2.0	Hard

2.0.0 - 2.0.2

The following table lists the dependencies for versions 2.0.0 through 2.0.2 of this component.

Dependency	Compatible versions	Dependency type
Client device auth	>=2.0.0 <2.1.0	Hard

For more information about component dependencies, see the component recipe reference.

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

`moquette`

(Optional) The Moquette MQTT broker configuration to use. You can configure a subset of Moqeutte configuration options in this component. For more information, see the inline comments in the Moquette configuration file.

This object contains the following information:

`ssl_port`

(Optional) The port where the MQTT broker operates.

> **Note**
>
> If you specify a different port, and you use the MQTT bridge component to relay MQTT messages to other brokers, you must use MQTT bridge v2.1.0 or later. Configure it to use the port where the MQTT broker operates.
> If you specify a different port, and you use the IP detector component to manage MQTT broker endpoints, you must use IP detector v2.1.0 or later. Configure it to report the port where the MQTT broker operates.

Default: 8883

`host`

(Optional) The interface where the MQTT broker binds. For example, you might change this parameter so that the MQTT broker binds only to a specific local network.

Default: `0.0.0.0` (binds to all network interfaces)

startupTimeoutSeconds

(Optional) The maximum of time in seconds for the component to start. The component's state changes to BROKEN if it exceeds this timeout.

Default: `120`

Example: Configuration merge update

The following example configuration specifies to operate the MQTT broker on port 443.

```
{
  "moquette": {
    "ssl_port": "443"
  }
}
```

Local log file

This component uses the same log file as the Greengrass nucleus component.

Linux

```
/greengrass/v2/logs/greengrass.log
```

Windows

```
C:\greengrass\v2\logs\greengrass.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
  ```

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.3.7	Version updated for client device auth version 2.5.0 release.
2.3.6	Bug fixes and improvements • General bug fixes and improvements.
2.3.5	Bug fixes and improvements • Updated Moquette to version 0.17.

Version	Changes
2.3.4	Bug fixes and improvements • Fixes an issue where clients may experience invalid session errors when sending or receiving messages, due to duplicate client IDs. This issue caused the client's session to close.
2.3.3	New features Adds a new `startupTimeoutSeconds` configuration option.
2.3.2	Version updated for client device auth version 2.4.0 release.
2.3.1	Bug fixes and improvements • Fixes a race condition where clients may be disconnected after attempting to reconnect, due to an invalid session.
2.3.0	Adds support for certificate chains.
2.2.0	Version updated for client device auth version 2.2.0 release.
2.1.0	Bug fixes and improvements • Updates this component to use Moquette version 0.16, which improves performance and includes several other improvements. • Fixes an issue where the local MQTT server certificate rotates more often than intended in certain scenarios. To apply this fix, you must also use v2.1.0 or later of the client device auth component.
2.0.2	Bug fixes and improvements • Increases the maximum MQTT message size from 8,092 bytes to 128 kilobytes. The effective MQTT message payload limit is slightly less, because the message size limit includes message headers. • Adds support for integer values in the `ssl_port` parameter.
2.0.1	Version updated for Greengrass nucleus version 2.4.0 release.

Version	Changes
2.0.0	Initial version.

MQTT 5 broker (EMQX)

The EMQX MQTT broker component (`aws.greengrass.clientdevices.mqtt.EMQX`) handles MQTT messages between client devices and a Greengrass core device. This component provides a modified version of the EMQX MQTT 5.0 broker. Deploy this MQTT broker to use MQTT 5 features in communication between client devices and a core device. For more information about how to choose an MQTT broker, see Choose an MQTT broker.

This broker implements the MQTT 5.0 protocol. It includes support for session and message expiration intervals, user properties, shared subscriptions, topic aliases, and more. MQTT 5 is backwards compatible with MQTT 3.1.1, so if you run the Moquette MQTT 3.1.1 broker, you can replace it with the EMQX MQTT 5 broker, and client devices can continue to connect and operate as usual.

> **Note**
>
> Client devices are local IoT devices that connect to a Greengrass core device to send MQTT messages and data to process. For more information, see Interact with local IoT devices.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Local log file
- Licenses
- Changelog

Versions

This component has the following versions:

- 2.0.x
- 1.2.x
- 1.1.x
- 1.0.x

Type

This component is a generic component (`aws.greengrass.generic`). The Greengrass nucleus runs the component's lifecycle scripts.

For more information, see Component types.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

Requirements

This component has the following requirements:

- The core device must be able to accept connections on the port where the MQTT broker operates. This component runs the MQTT broker on port 8883 by default. You can specify a different port when you configure this component.

  If you specify a different port, and you use the MQTT bridge component to relay MQTT messages to other brokers, you must use MQTT bridge v2.1.0 or later. Configure it to use the port where the MQTT broker operates.

  If you specify a different port, and you use the IP detector component to manage MQTT broker endpoints, you must use IP detector v2.1.0 or later. Configure it to report the port where the MQTT broker operates.

- On Linux core devices, Docker installed and configured on the core device:
  - Docker Engine 1.9.1 or later installed on the Greengrass core device. Versions up to and including 29.6.0, such as 20.10, 24.0, 25.0, and 27.5, work with the AWS IoT Greengrass Core software. You must install Docker directly on the core device before you deploy components that run Docker containers.
  - The Docker daemon started and running on the core device before you deploy this component.
  - The system user that runs this component must have root or administrator permissions. Alternatively, you can run this component as a system user in the `docker` group and configure this component's `requiresPrivileges` option to `false` to run the EQMX MQTT broker without privileges.
- The EMQX MQTT broker component is supported to run in a VPC.
- The EMQX MQTT broker component is not supported on the `armv7` platform.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

2.0.2 – 2.0.3

The following table lists the dependencies for versions 2.0.2 and 2.0.3 of this component.

Dependency	Compatible versions	Dependency type
Client device auth	>=2.2.0 <2.6.0	Soft

2.0.1

The following table lists the dependencies for version 2.0.1 of this component.

Dependency	Compatible versions	Dependency type
Client device auth	>=2.2.0 <2.6.0	Hard

2.0.0

The following table lists the dependencies for version 2.0.0 of this component.

Dependency	Compatible versions	Dependency type
Client device auth	>=2.2.0 <2.5.0	Hard

1.2.2 – 1.2.3

The following table lists the dependencies for versions 1.2.2 to 1.2.3 of this component.

Dependency	Compatible versions	Dependency type
Client device auth	>=2.2.0 <2.5.0	Hard

1.2.0 and 1.2.1

The following table lists the dependencies for versions 1.2.0 and 1.2.1 of this component.

Dependency	Compatible versions	Dependency type
Client device auth	>=2.2.0 <2.4.0	Hard

1.0.0 and 1.1.0

The following table lists the dependencies for versions 1.0.0 and 1.1.0 of this component.

Dependency	Compatible versions	Dependency type
Client device auth	>=2.2.0 <2.3.0	Hard

For more information about component dependencies, see the component recipe reference.

Configuration

2.0.0 - 2.0.3

This component provides the following configuration parameters that you can customize when you deploy the component.

> **Important**
>
> If you use version 2 of the MQTT 5 broker (EMQX) component, you must update your configuration file. Version 1 configuration files do not work with version 2.

emqxConfig

(Optional) The EMQX MQTT broker configuration to use. You can set EMQX configuration options in this component.

When you use the EMQX broker, Greengrass uses a default configuration. This configuration is used unless you modify it using this field.

Modifying the following configuration settings causes the EMQX broker component to restart. Other configuration changes apply without restarting the component.

- `emqxConfig/cluster`
- `emqxConfig/node`
- `emqxConfig/rpc`

> **Note**
>
> `aws.greengrass.clientdevices.mqtt.EMQX` allows you to configure security-sensitive options. These include TLS settings, authentication, and authorization providers. We recommended the default configuration that uses mutual TLS authentication and the Greengrass client device auth provider.

Example: Default configuration

The following example shows the defaults set for the MQTT 5 (EMQX) broker. You can override these settings using the `emqxConfig` configuration setting.

```
{
  "authorization": {
    "no_match": "deny",
    "sources": []
  },
  "node": {
    "cookie": "<placeholder>"
  },
  "listeners": {
     "ssl": {
       "default": {
         "ssl_options": {
           "keyfile": "{work:path}\\data\\key.pem",
           "certfile": "{work:path}\\data\\cert.pem",
           "cacertfile": null,
           "verify": "verify_peer",
           "versions": ["tlsv1.3", "tlsv1.2"],
           "fail_if_no_peer_cert": true
         }
       }
     },
     "tcp": {
       "default": {
         "enabled": false
       }
     },
     "ws": {
       "default": {
         "enabled": false
       }
     },
     "wss": {
       "default": {
         "enabled": false
       }
     }
  },
  "plugins": {
    "states": [{"name_vsn": "gg-1.0.0", "enable": true}],
```

```
    "install_dir": "plugins"
  }
}
```

authMode

(Optional) Sets the authorization provider for the broker. Can be one of the following values:

- `enabled` – (Default) Use the Greengrass authentication and authorization provider.
- `bypass_on_failure` – Use the Greengrass authentication provider, then use any remaining authentication providers in the EMQX provider chain if Greengrass denies either authentication or authorization.
- `bypass` – The Greengrass provider is disabled. Authentication and authorization is handled by the EMQX provider chain.

`requiresPrivilege`

(Optional) On Linux core devices, you can specify to run the EMQX MQTT broker without root or administrator privileges. If you set this option to `false`, the system user that runs this component must be a member of the `docker` group.

Default: `true`

`startupTimeoutSeconds`

(Optional) The maximum of time in seconds for the EMQX MQTT broker to start. The component's state changes to BROKEN if it exceeds this timeout.

Default: `90`

`ipcTimeoutSeconds`

(Optional) The maximum of time in seconds for the component to wait for the Greengrass nucleus to respond to interprocess communication (IPC) requests. Increase this number if this component reports timeout errors when it checks if a client device is authorized.

Default: `5`

`crtLogLevel`

(Optional) The log level for the AWS Common Runtime (CRT) library.

Defaults to the EMQX MQTT broker log level (`log.level` in `emqx`).

`restartIdentifier`

(Optional) Configure this option to restart the EMQX MQTT broker. When this configuration value changes, this component restarts the MQTT broker. You can use this option to force client devices to disconnect.

`dockerOptions`

(Optional) Configure this option only on Linux operating systems to add parameters to the Docker command line. For example, to map additional ports, use the `-p` Docker parameter:

```
"-p 1883:1883"
```

Example: Updating a v1.x configuration file to v2.x

The following example shows the changes necessary to update a v1.x configuration file to version 2.x.

The version 1.x configuration file:

```
{
    "emqx": {
        "listener.ssl.external": "443",
        "listener.ssl.external.max_connections": "1024000",
        "listener.ssl.external.max_conn_rate": "500",
        "listener.ssl.external.rate_limit": "50KB,5s",
        "listener.ssl.external.handshake_timeout": "15s",
        "log.level": "warning"
    },
    "mergeConfigurationFiles": {
        "etc/plugins/aws_greengrass_emqx_auth.conf": "auth_mode=enabled\n
 use_greengrass_managed_certificates=true\n"
    }
}
```

The equivalent configuration file for v2:

```
{
    "emqxConfig": {
```

```
        "listeners": {
            "ssl": {
                "default": {
                   "bind": "8883",
                   "max_connections": "1024000",
                   "max_conn_rate": "500",
                   "ssl_options": {
                        "handshake_timeout": "15s"
                   }
                }
            }
        },
        "log": {
            "console": {
              "enable": true,
              "level": "warning"
            }
        }
    },
    "authMode": "enabled"
}
```

There is no equivalent to the `listener.ssl.external.rate_limit` configuration entry. The `use_greengrass_managed_certificates` configuration option has been removed.

Example: Set a new port for the broker

The following example changes the port where the MQTT broker operates from the default 8883 to port 1234. If you are using Linux, include the `dockerOptions` field.

```
{
  "emqxConfig": {
    "listeners": {
      "ssl": {
        "default": {
          "bind": 1234
        }
      }
    }
  },
  "dockerOptions": "-p 1234:1234"
}
```

Example: Adjust the MQTT broker's log level

The following example changes the MQTT broker's log level to `debug`. You can choose from the following log levels:

- `debug`
- `info`
- `notice`
- `warning`
- `error`
- `critical`
- `alert`
- `emergency`

The default log level is `warning`.

```
{
  "emqxConfig": {
    "log": {
      "console": {
         "level": "debug"
      }
    }
  }
}
```

Example: Enable the EMQX dashboard

The following example enables the EMQX dashboard so that you can monitor and manage your broker. If you are using Linux, include the `dockerOptions` field.

```
{
  "emqxConfig": {
    "dashboard": {
      "listeners": {
        "http": {
          "bind": 18083
        }
      }
```

```
    }
  },
  "dockerOptions": "-p 18083:18083"
}
```

1.0.0 - 1.2.2

This component provides the following configuration parameters that you can customize when you deploy the component.

`emqx`

(Optional) The EMQX MQTT broker configuration to use. You can configure a subset of EMQX configuration options in this component.

This object contains the following information:

`listener.ssl.external`

(Optional) The port where the MQTT broker operates.

> **Note**
>
> If you specify a different port, and you use the MQTT bridge component to relay MQTT messages to other brokers, you must use MQTT bridge v2.1.0 or later. Configure it to use the port where the MQTT broker operates.
> If you specify a different port, and you use the IP detector component to manage MQTT broker endpoints, you must use IP detector v2.1.0 or later. Configure it to report the port where the MQTT broker operates.

Default: `8883`

`listener.ssl.external.max_connections`

(Optional) The maximum number of concurrent connections that the MQTT broker supports.

Default: `1024000`

`listener.ssl.external.max_conn_rate`

(Optional) The maximum number of new connections per second the MQTT broker can receive.

Default: `500`

`listener.ssl.external.rate_limit`

(Optional) The bandwidth limit for all connections to the MQTT broker. Specify the bandwidth and duration for that bandwidth separated by a comma (`,`) in the following format: `bandwidth,duration`. For example, you can specify `50KB,5s` to limit the MQTT broker to 50 kilobytes (KB) of data every 5 seconds.

`listener.ssl.external.handshake_timeout`

(Optional) The amount of time that the MQTT broker waits to finish authenticating a new connection.

Default: `15s`

`mqtt.max_packet_size`

(Optional) The maximum size of an MQTT message.

Default: `268435455` (256 MB minus 1)

`log.level`

(Optional) The log level for the MQTT broker. Choose from the following options:

- `debug`
- `info`
- `notice`
- `warning`
- `error`
- `critical`
- `alert`
- `emergency`

The default log level is `warning`.

`requiresPrivilege`

(Optional) On Linux core devices, you can specify to run the EMQX MQTT broker without root or administrator privileges. If you set this option to `false`, the system user that runs this component must be a member of the `docker` group.

Default: `true`

`startupTimeoutSeconds`

(Optional) The maximum of time in seconds for the EMQX MQTT broker to start. The component's state changes to `BROKEN` if it exceeds this timeout.

Default: `90`

`ipcTimeoutSeconds`

(Optional) The maximum of time in seconds for the component to wait for the Greengrass nucleus to respond to interprocess communication (IPC) requests. Increase this number if this component reports timeout errors when it checks if a client device is authorized.

Default: `5`

`crtLogLevel`

(Optional) The log level for the AWS Common Runtime (CRT) library.

Defaults to the EMQX MQTT broker log level (`log.level` in `emqx`).

`restartIdentifier`

(Optional) Configure this option to restart the EMQX MQTT broker. When this configuration value changes, this component restarts the MQTT broker. You can use this option to force client devices to disconnect.

`dockerOptions`

(Optional) Configure this option only on Linux operating systems to add parameters to the Docker command line. For example, to map additional ports, use the `-p` Docker parameter:

```
"-p 1883:1883"
```

`mergeConfigurationFiles`

(Optional) Configure this option to add to or override the defaults in the specified EMQX configuration files. For information about the configuration files and their formats, see Configuration in the *EMQX 4.0 Documentation*. The values that you specify are appended to the configuration file.

The following example updates the `etc/emqx.conf` file.

```
"mergeConfigurationFiles": {
    "etc/emqx.conf": "broker.sys_interval=30s\nbroker.sys_heartbeat=10s"
```

```
},
```

In addition to the configuration files supported by EMQX, Greengrass supports a file that configures the Greengrass auth plugin for EMQX called `etc/plugins/aws_greengrass_emqx_auth.conf`. There are two supported options, `auth_mode` and `use_greengrass_managed_certificates`. To use another auth provider, set the `auth_mode` option to one of the following:

- `enabled` – (Default) Use the Greengrass authentication and authorization provider.
- `bypass_on_failure` – Use the Greengrass authentication provider, then use any remaining authentication providers in the EMQX provider chain if Greengrass denies either authentication or authorization.
- `bypass` – The Greengrass provider is disabled. Authentication and authorization is then handled by the EMQX provider chain.

If the `use_greengrass_managed_certificates` is `true`, this option indicates that Greengrass manages the broker TLS certificates. If `false`, it indicates that you provide the certificates through another source.

The following example updates the defaults in the `etc/plugins/aws_greengrass_emqx_auth.conf` configuration file.

```
"mergeConfigurationFiles": {
    "etc/plugins/aws_greengrass_emqx_auth.conf": "auth_mode=enabled\n
 use_greengrass_managed_certificates=true\n"
  },
```

> **Note**
>
> `aws.greengrass.clientdevices.mqtt.EMQX` allows you to configure security-sensitive options. These include TLS settings, authentication, and authorization providers. The recommended configuration is the default configuration that uses mutual TLS authentication and the Greengrass Client Device Auth provider.

`replaceConfigurationFiles`

(Optional) Configure this option to replace the specified EMQX configuration files. The values that you specify replace the entire existing configuration file. You can't specify the

`etc/emqx.conf` file in this section. You must use `mergeConfigurationFile` to modify `etc/emqx.conf`.

Example: Configuration merge update

The following example configuration specifies to operate the MQTT broker on port 443.

```
{
  "emqx": {
    "listener.ssl.external": "443",
    "listener.ssl.external.max_connections": "1024000",
    "listener.ssl.external.max_conn_rate": "500",
    "listener.ssl.external.rate_limit": "50KB,5s",
    "listener.ssl.external.handshake_timeout": "15s",
    "log.level": "warning"
  },
  "requiresPrivilege": "true",
  "startupTimeoutSeconds": "90",
  "ipcTimeoutSeconds": "5"
}
```

Local log file

This component uses the following log file.

Linux

```
/greengrass/v2/logs/aws.greengrass.clientdevices.mqtt.EMQX.log
```

Windows

```
C:\greengrass\v2\logs\aws.greengrass.clientdevices.mqtt.EMQX.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

Linux

```
sudo tail -f /greengrass/v2/logs/aws.greengrass.clientdevices.mqtt.EMQX.log
```

Windows (PowerShell)

```
Get-Content C:\greengrass\v2\logs\aws.greengrass.clientdevices.mqtt.EMQX.log -
Tail 10 -Wait
```

Licenses

On Windows operating systems, this software includes code distributed under the Microsoft Software License Terms - Microsoft Visual Studio Community 2022. By downloading this software, you agree to that code's license terms.

This component is released under the Greengrass Core Software License Agreement.

Changelog

The following table describes the changes in each version of the component.

v2.x

Version	Changes
2.0.3	Bug fixes and improvements • Fixes an issue where EMQX doesn't start on Windows if the path contains spaces.
2.0.2	Bug fixes and improvements • Fixes an issue where EMQX starts up before the Client device auth component is ready.
2.0.1	Version updated for client device auth version 2.5.0 release.
2.0.0	This version of the MQTT 5 broker (EMQX) expects different configuration parameters than version 1.x. If you use a non-default configuration for

Version	Changes
	version 1.x, you must update the component's configuration for 2.x. For more information, see Configuration. New features • Upgrades the MQTT broker to EMQX 5.1.1. • Enables broker configuration changes without restarting the component. Updates • Adds a new `emqxConfig` configuration field that replaces the `emqx`, `mergeConfigurationFiles` , and `replaceConfigurationFiles` configuration fields.

v1.x

Version	Changes
1.2.3	Bug fixes and improvements • Fixes an issue where clients couldn't interact with EMQX after previously authenticating by disconnecting and reauthenticating the client.
1.2.2	Version updated for client device auth version 2.4.0 release.
1.2.1	Bug fixes and improvements • Fixes an issue where the component won't startup on Windows if Visual C++ Redistributable is not already present. • Updates EMQX to version 4.4.14.
1.2.0	Adds support for certificate chains.
1.1.0	New features • Adds support for EMQX configurations including broker options and plug-ins.

Version	Changes
	Bug fixes and improvements • Updates EMQX to version 4.4.9.
1.0.1	Fixes an issue during the TLS handshake which results in some MQTT clients failing to connect.
1.0.0	Initial version.

Nucleus telemetry emitter

The nucleus telemetry emitter component (`aws.greengrass.telemetry.NucleusEmitter`) gathers system health telemetry data and publishes it continually to a local topic and an AWS IoT Core MQTT topic. This component enables you to gather real-time system telemetry on your Greengrass core devices. For information about the Greengrass telemetry agent that publishes system telemetry data to Amazon EventBridge, see Gather system health telemetry data from AWS IoT Greengrass core devices.

By default, the nucleus telemetry emitter component publishes telemetry data every 60 seconds to the following local publish/subscribe topic.

```
$local/greengrass/telemetry
```

The nucleus telemetry emitter component doesn't publish to an AWS IoT Core MQTT topic by default. You can configure this component to publish to an AWS IoT Core MQTT topic when you deploy it. The use of an MQTT topic to publish data to the AWS Cloud is subject to AWS IoT Core pricing.

AWS IoT Greengrass provides several community components to help you analyze and visualize telemetry data locally on your core device using InfluxDB and Grafana. These components use telemetry data from the nucleus emitter component. For more information, see the README for the InfluxDB publisher component.

Topics

- Versions
- Type

- Operating system
- Dependencies
- Configuration
- Output data
- Usage
- Local log file
- Changelog

Versions

This component has the following versions:

- 1.0.x

Type

This component is a plugin component (`aws.greengrass.plugin`). The Greengrass nucleus runs this component in the same Java Virtual Machine (JVM) as the nucleus. The nucleus restarts when you change this component's version on the core device.

This component uses the same log file as the Greengrass nucleus. For more information, see Monitor AWS IoT Greengrass logs.

For more information, see Component types.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for

the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

1.0.12

The following table lists the dependencies for version 1.0.12 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.4.0 <2.17.0	Soft

1.0.11

The following table lists the dependencies for version 1.0.11 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.4.0 <2.16.0	Hard

1.0.10

The following table lists the dependencies for version 1.0.10 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.4.0 <2.15.0	Hard

1.0.9

The following table lists the dependencies for version 1.0.9 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.4.0 <2.14.0	Hard

1.0.8

The following table lists the dependencies for version 1.0.8 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.4.0 <2.13.0	Hard

1.0.7

The following table lists the dependencies for version 1.0.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.4.0 <2.12.0	Hard

1.0.6

The following table lists the dependencies for version 1.0.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.4.0 <2.11.0	Hard

1.0.5

The following table lists the dependencies for version 1.0.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.4.0 <2.10.0	Hard

1.0.4

The following table lists the dependencies for version 1.0.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.4.0 <2.9.0	Hard

1.0.3

The following table lists the dependencies for version 1.0.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.4.0 <2.8.0	Hard

1.0.2

The following table lists the dependencies for version 1.0.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.4.0 <2.7.0	Hard

1.0.1

The following table lists the dependencies for version 1.0.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.4.0 <2.6.0	Hard

1.0.0

The following table lists the dependencies for version 1.0.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.4.0 <2.5.0	Hard

For more information about component dependencies, see the component recipe reference.

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

`pubSubPublish`

(Optional) Defines whether to publish telemetry data to the `$local/greengrass/telemetry` topic. Supported values are `true` and `false`.

Default: `true`

`mqttTopic`

(Optional) The AWS IoT Core MQTT topic to which this component publishes telemetry data.

Set this value to the AWS IoT Core MQTT topic to which you want to publish telemetry data. When this value is empty, the nucleus emitter doesn't publish telemetry data to the AWS Cloud.

> **Note**
>
> The use of an MQTT topic to publish data to the AWS Cloud is subject to AWS IoT Core pricing.

Default: `""`

`telemetryPublishIntervalMs`

(Optional) The amount of time (in milliseconds) between which the component publishes telemetry data. If you set this value lower than the minimum supported value, the component uses the minimum value instead.

> **Note**
>
> Lower publish intervals result in higher CPU usage on your core device. We recommend that you start with the default publish interval and adjust it based on your device's CPU usage.

Minimum: `500`

Default: `60000`

Example: Configuration merge update

The following example shows a sample configuration merge update that enables publishing telemetry data every 5 seconds to the `$local/greengrass/telemetry` topic and the `greengrass/myTelemetry` AWS IoT Core MQTT topic.

```
{
  "pubSubPublish": "true",
  "mqttTopic": "greengrass/myTelemetry",
  "telemetryPublishIntervalMs": 5000
}
```

Output data

This component publishes telemetry metrics as a JSON array on the following topic.

Local topic: `$local/greengrass/telemetry`

You can optionally choose to also publish telemetry metrics to an AWS IoT Core MQTT topic. For more information about topics, see MQTT topics in the *AWS IoT Core Developer Guide*.

Example data

```
[
  {
    "A": "Average",
    "N": "CpuUsage",
    "NS": "SystemMetrics",
    "TS": 1627597331445,
    "U": "Percent",
    "V": 26.21981271562346
  },
  {
    "A": "Count",
    "N": "TotalNumberOfFDs",
    "NS": "SystemMetrics",
    "TS": 1627597331445,
    "U": "Count",
    "V": 7316
  },
```

```
  {
    "A": "Count",
    "N": "SystemMemUsage",
    "NS": "SystemMetrics",
    "TS": 1627597331445,
    "U": "Megabytes",
    "V": 10098
  },
  {
    "A": "Count",
    "N": "NumberOfComponentsStarting",
    "NS": "GreengrassComponents",
    "TS": 1627597331446,
    "U": "Count",
    "V": 0
  },
  {
    "A": "Count",
    "N": "NumberOfComponentsInstalled",
    "NS": "GreengrassComponents",
    "TS": 1627597331446,
    "U": "Count",
    "V": 0
  },
  {
    "A": "Count",
    "N": "NumberOfComponentsStateless",
    "NS": "GreengrassComponents",
    "TS": 1627597331446,
    "U": "Count",
    "V": 0
  },
  {
    "A": "Count",
    "N": "NumberOfComponentsStopping",
    "NS": "GreengrassComponents",
    "TS": 1627597331446,
    "U": "Count",
    "V": 0
  },
  {
    "A": "Count",
    "N": "NumberOfComponentsBroken",
    "NS": "GreengrassComponents",
```

```
    "TS": 1627597331446,
    "U": "Count",
    "V": 0
  },
  {
    "A": "Count",
    "N": "NumberOfComponentsRunning",
    "NS": "GreengrassComponents",
    "TS": 1627597331446,
    "U": "Count",
    "V": 7
  },
  {
    "A": "Count",
    "N": "NumberOfComponentsErrored",
    "NS": "GreengrassComponents",
    "TS": 1627597331446,
    "U": "Count",
    "V": 0
  },
  {
    "A": "Count",
    "N": "NumberOfComponentsNew",
    "NS": "GreengrassComponents",
    "TS": 1627597331446,
    "U": "Count",
    "V": 0
  },
  {
    "A": "Count",
    "N": "NumberOfComponentsFinished",
    "NS": "GreengrassComponents",
    "TS": 1627597331446,
    "U": "Count",
    "V": 2
  }
]
```

The output array contains a list of metrics that have the following properties:

A

The aggregation type for the metric.

For the `CpuUsage` metric, this property is set to `Average` because the published value of the metric is the average CPU usage amount since the last publish event.

For all other metrics, the nucleus emitter doesn't aggregate the metric value, and this property is set to `Count`.

`N`

The name of the metric.

`NS`

The metric namespace.

`TS`

The timestamp of when the data was gathered.

`U`

The unit of the metric value.

`V`

The metric value.

The nucleus emitter publishes the following metrics:

Name	Description	
System		
`SystemMemUsage`	The amount of memory currently in use by all applications on the Greengrass core device, including the operating system.	
`CpuUsage`	The amount of CPU currently in use by all applications on the Greengrass core device, including the operating system.	

Name	Description
TotalNumberOfFDs	The number of file descriptors stored by the operating system of the Greengrass core device. One file descriptor uniquely identifies one open file.
Greengrass nucleus	
NumberOfComponentsRunning	The number of components that are running on the Greengrass core device.
NumberOfComponentsErrored	The number of components that are in error state on the Greengrass core device.
NumberOfComponentsInstalled	The number of components that are installed on the Greengrass core device.
NumberOfComponentsStarting	The number of components that are starting on the Greengrass core device.
NumberOfComponentsNew	The number of components that are new on the Greengrass core device.
NumberOfComponentsStopping	The number of components that are stopping on the Greengrass core device.
NumberOfComponentsFinished	The number of components that are finished on the Greengrass core device.

Name	Description	
`NumberOfComponentsBroken`	The number of components that are broken on the Greengrass core device.	
`NumberOfComponentsStateless`	The number of components that are stateless on the Greengrass core device.	

Usage

To use system health telemetry data, you can create custom components that subscribe to the topics to which the nucleus emitter publishes the telemetry data, and react to that data as needed. Because the nucleus emitter component provides the option to publish telemetry data to a local topic, you can subscribe to that topic, and use the published data to act locally on your core device. The core device can then react to telemetry data even when it has limited connectivity to the cloud.

For example, you can configure a component that listens on the `$local/greengrass/telemetry` topic for telemetry data and send the data to the stream manager component to stream your data to the AWS Cloud. For more information about creating such a component, see Publish/subscribe local messages and Create custom components that use stream manager.

Local log file

This component uses the same log file as the Greengrass nucleus component.

Linux

```
/greengrass/v2/logs/greengrass.log
```

Windows

```
C:\greengrass\v2\logs\greengrass.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
  ```

Changelog

The following table describes the changes in each version of the component.

Version	Changes
1.0.14	Updates the component version for the Greengrass nucleus version 2.18.0 release.
1.0.13	Updates the component version for the Greengrass nucleus version 2.17.0 release.
1.0.12	Version updated for Greengrass nucleus version 2.16.0 release.
1.0.11	Version updated for Greengrass nucleus version 2.15.0 release.
1.0.10	Version updated for Greengrass nucleus version 2.14.0 release.
1.0.9	Version updated for Greengrass nucleus version 2.13.0 release.
1.0.8	Version updated for Greengrass nucleus version 2.12.0 release.
1.0.7	Version updated for Greengrass nucleus version 2.11.0 release.
1.0.6	Version updated for Greengrass nucleus version 2.10.0 release.

Version	Changes
1.0.5	Version updated for Greengrass nucleus version 2.9.0 release.
1.0.4	Version updated for Greengrass nucleus version 2.8.0 release.
1.0.3	Version updated for Greengrass nucleus version 2.7.0 release.
1.0.2	Version updated for Greengrass nucleus version 2.6.0 release.
1.0.1	Version updated for Greengrass nucleus version 2.5.0 release.
1.0.0	Initial version.

PKCS#11 provider

The PKCS#11 provider component (`aws.greengrass.crypto.Pkcs11Provider`) enables you to configure the AWS IoT Greengrass Core software to use a hardware security module (HSM) through the PKCS#11 interface. This component enables you to securely store certificate and private key files so that they aren't exposed or duplicated in software. For more information, see Hardware security integration.

> **Note**
>
> This component is required for Greengrass nucleus only. Greengrass nucleus lite v2.5.0 and later includes built-in PKCS#11 support and does not require this component. For more information, see PKCS#11 support in the AWS IoT Greengrass nucleus lite GitHub repository.

To provision a Greengrass core device that stores its certificate and private key in an HSM, you must specify this component as a provisioning plugin when you install the AWS IoT Greengrass Core software. For more information, see Install AWS IoT Greengrass Core software with manual resource provisioning.

AWS IoT Greengrass provides this component as JAR file that you can download to specify as a provisioning plugin during installation. You can download the latest version of the component's

JAR file as the following URL: https://d2s8p88vqu9w66.cloudfront.net/releases/Pkcs11Provider/aws.greengrass.crypto.Pkcs11Provider-latest.jar.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Local log file
- Changelog

Versions

This component has the following versions:

- 2.0.x

Type

This component is a plugin component (`aws.greengrass.plugin`). The Greengrass nucleus runs this component in the same Java Virtual Machine (JVM) as the nucleus. The nucleus restarts when you change this component's version on the core device.

This component uses the same log file as the Greengrass nucleus. For more information, see Monitor AWS IoT Greengrass logs.

For more information, see Component types.

Operating system

This component can be installed on Linux core devices only.

Requirements

This component has the following requirements:

- A hardware security module that supports the PKCS#1 v1.5 signature scheme and RSA keys with an RSA-2048 key size (or larger) or ECC keys.

  > **Note**
  >
  > To use a hardware security module with ECC keys, you must use Greengrass nucleus v2.5.6 or later.
  > To use a hardware security module and secret manager, you must use a hardware security module with RSA keys.

- A PKCS#11 provider library that the AWS IoT Greengrass Core software can load at runtime (using libdl) to invoke PKCS#11 functions. The PKCS#11 provider library must implement the following PKCS#11 API operations:
  - `C_Initialize`
  - `C_Finalize`
  - `C_GetSlotList`
  - `C_GetSlotInfo`
  - `C_GetTokenInfo`
  - `C_OpenSession`
  - `C_GetSessionInfo`
  - `C_CloseSession`
  - `C_Login`
  - `C_Logout`
  - `C_GetAttributeValue`
  - `C_FindObjectsInit`
  - `C_FindObjects`
  - `C_FindObjectsFinal`
  - `C_DecryptInit`
  - `C_Decrypt`
  - `C_DecryptUpdate`
  - `C_DecryptFinal`
  - `C_SignInit`
  - `C_Sign`

  - `C_SignUpdate`
  - `C_SignFinal`
  - `C_GetMechanismList`
  - `C_GetMechanismInfo`
  - `C_GetInfo`
  - `C_GetFunctionList`
- The hardware module must be resolvable by slot label, as defined in the PKCS#11 specification.
- You must store the private key and certificate in the HSM in the same slot, and they must use the same object label and object ID, if the HSM supports object IDs.
- The certificate and private key must be resolvable by object labels.
- The private key must have the following permissions:
  - `sign`
  - `decrypt`
- (Optional) To use the secret manager component, you must use version 2.1.0 or later, and the private key must have the following permissions:
  - `unwrap`
  - `wrap`
- (Optional) If you are using the TPM2 library and running the Greengrass core as a service, you must provide an environment variable with the location of the PKCS#11 store. The following example is a systemd service file with the required environment variable:

```
[Unit]
Description=Greengrass Core
After=network.target

[Service]
Type=simple
PIDFile=/var/run/greengrass.pid
Environment=TPM2_PKCS11_STORE=/path/to/store/directory
RemainAfterExit=no
Restart=on-failure
RestartSec=10
ExecStart=/bin/sh /greengrass/v2/alts/current/distro/bin/loader

[Install]
```

```
WantedBy=multi-user.target
```

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

2.0.11

The following table lists the dependencies for version 2.0.11 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.3 <2.17.0	Soft

2.0.10

The following table lists the dependencies for version 2.0.10 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.3 <2.16.0	Soft

2.0.9

The following table lists the dependencies for version 2.0.9 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.3 <2.15.0	Soft

2.0.8

The following table lists the dependencies for version 2.0.8 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.3 <2.14.0	Soft

2.0.7

The following table lists the dependencies for version 2.0.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.3 <2.13.0	Soft

2.0.6

The following table lists the dependencies for version 2.0.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.3 <2.12.0	Soft

2.0.5

The following table lists the dependencies for version 2.0.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.3 <2.11.0	Soft

2.0.4

The following table lists the dependencies for version 2.0.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.3 <2.10.0	Soft

2.0.3

The following table lists the dependencies for version 2.0.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.3 <2.9.0	Soft

2.0.2

The following table lists the dependencies for version 2.0.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.3 <2.8.0	Soft

2.0.1

The following table lists the dependencies for version 2.0.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.3 <2.7.0	Soft

2.0.0

The following table lists the dependencies for version 2.0.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.3 <2.6.0	Soft

For more information about component dependencies, see the component recipe reference.

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

`name`

A name for the PKCS#11 configuration.

`library`

The absolute file path to the PKCS#11 implementation's library that the AWS IoT Greengrass Core software can load with libdl.

`slot`

The ID of the slot that contains the private key and device certificate. This value is different than the slot index or slot label.

`userPin`

The user PIN to use to access the slot.

Example: Configuration merge update

```
{
  "name": "softhsm_pkcs11",
  "library": "/usr/lib/softhsm/libsofthsm2.so",
  "slot": 1,
  "userPin": "1234"
}
```

Local log file

This component uses the same log file as the Greengrass nucleus component.

Linux

```
/greengrass/v2/logs/greengrass.log
```

Windows

```
C:\greengrass\v2\logs\greengrass.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
  ```

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.0.13	Updates the component version for the Greengrass nucleus version 2.18.0 release.
2.0.12	Updates the component version for the Greengrass nucleus version 2.17.0 release.
2.0.11	Version updated for Greengrass nucleus version 2.16.0 release.
2.0.10	Version updated for Greengrass nucleus version 2.15.0 release.
2.0.9	Version updated for Greengrass nucleus version 2.14.0 release.
2.0.8	Version updated for Greengrass nucleus version 2.13.0 release.

Version	Changes
2.0.7	Version updated for Greengrass nucleus version 2.12.0 release.
2.0.6	Version updated for Greengrass nucleus version 2.11.0 release.
2.0.5	Version updated for Greengrass nucleus version 2.10.0 release.
2.0.4	Version updated for Greengrass nucleus version 2.9.0 release.
2.0.3	Version updated for Greengrass nucleus version 2.8.0 release.
2.0.2	Version updated for Greengrass nucleus version 2.7.0 release.
2.0.1	Version updated for Greengrass nucleus version 2.6.0 release.
2.0.0	Initial version.

Secret manager

The secret manager component (`aws.greengrass.SecretManager`) deploys secrets from AWS Secrets Manager to Greengrass core devices. Use this component to securely use credentials, such as passwords, in custom components on your Greengrass core devices. For more information about Secrets Manager, see What is AWS Secrets Manager? in the *AWS Secrets Manager User Guide*.

To access this component's secrets in your custom Greengrass components, use the GetSecretValue operation in the AWS IoT Device SDK. For more information, see Use the AWS IoT Device SDK to communicate with the Greengrass nucleus, other components, and AWS IoT Core and Retrieve secret values.

This component encrypts secrets on the core device to keep your credentials and passwords secure until you need to use them. It uses the core device's private key to encrypt and decrypt secrets.

Topics

- Versions
- Type
- Operating system
- Requirements

- Dependencies
- Configuration
- Local log file
- Changelog

Versions

This component has the following versions:

- 2.2.x
- 2.1.x
- 2.0.x

Type

This component is a plugin component (`aws.greengrass.plugin`). The Greengrass nucleus runs this component in the same Java Virtual Machine (JVM) as the nucleus. The nucleus restarts when you change this component's version on the core device.

This component uses the same log file as the Greengrass nucleus. For more information, see Monitor AWS IoT Greengrass logs.

For more information, see Component types.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

Requirements

This component has the following requirements:

- The Greengrass device role must allow the `secretsmanager:GetSecretValue` action, as shown in the following example IAM policy.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Action": [
        "secretsmanager:GetSecretValue"
      ],
      "Effect": "Allow",
      "Resource": [
      "arn:aws:secretsmanager:us-east-1:123456789012:secret:MySecret"
      ]
    }
  ]
}
```

Note

If you use a customer-managed AWS Key Management Service key to encrypt secrets, the device role must also allow the `kms:Decrypt` action.

For more information about IAM policies for Secrets Manager, see the following in the *AWS Secrets Manager User Guide*:

- Authentication and access control for AWS Secrets Manager
- Actions, resources, and context keys you can use in an IAM policy or secret policy for AWS Secrets Manager

- Custom components must define an authorization policy that allows `aws.greengrass#GetSecretValue` to get secrets that you store with this component. In this authorization policy, you can restrict components' access to specific secrets. For more information, see secret manager IPC authorization.
- (Optional) If you store the core device's private key and certificate in a hardware security module (HSM), the HSM must support RSA keys, the private key must have the `unwrap` permission, and the public key must have the `wrap` permission.

Endpoints and ports

This component must be able to perform outbound requests to the following endpoints and ports, in addition to endpoints and ports required for basic operation. For more information, see Allow device traffic through a proxy or firewall.

Endpoint	Port	Required	Description
`secretsmanager.` *`region`*`.amazonaws.com`	443	Yes	Download secrets to the core device.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

2.2.9 – 2.2.10

The following table lists the dependencies for versions 2.2.9 and 2.2.10 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.19.0	Soft

2.2.8

The following table lists the dependencies for version 2.2.8 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.18.0	Soft

2.2.7

The following table lists the dependencies for version 2.2.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.17.0	Soft

2.2.6

The following table lists the dependencies for version 2.2.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.16.0	Soft

2.2.2 – 2.2.5

The following table lists the dependencies for versions 2.2.2 through 2.2.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.15.0	Soft

2.2.0

The following table lists the dependencies for versions 2.2.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.13.0 <2.14.0	Soft

2.1.7 – 2.1.8

The following table lists the dependencies for versions 2.1.7 and 2.1.8 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.13.0	Soft

2.1.6

The following table lists the dependencies for version 2.1.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.12.0	Soft

2.1.5

The following table lists the dependencies for version 2.1.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.11.0	Soft

2.1.4

The following table lists the dependencies for version 2.1.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.10.0	Soft

2.1.3

The following table lists the dependencies for version 2.1.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.9.0	Soft

2.1.2

The following table lists the dependencies for version 2.1.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.8.0	Soft

2.1.1

The following table lists the dependencies for version 2.1.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.7.0	Soft

2.1.0

The following table lists the dependencies for version 2.1.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.6.0	Soft

2.0.9

The following table lists the dependencies for version 2.0.9 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.5.0	Soft

2.0.8

The following table lists the dependencies for version 2.0.8 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.4.0	Soft

2.0.7

The following table lists the dependencies for version 2.0.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.3.0	Soft

2.0.6

The following table lists the dependencies for version 2.0.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.2.0	Soft

2.0.4 and 2.0.5

The following table lists the dependencies for versions 2.0.4 and 2.0.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.3 <2.1.0	Soft

For more information about component dependencies, see the component recipe reference.

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

`periodicRefreshIntervalMin` (optional)

The interval in minutes where this component syncs the configured secrets on the core device with the latest secret values from the AWS Secrets Manager service. If this interval is not configured, secret manager will not refresh the configured secrets periodically.

```
{
  "cloudSecrets": [
    {
      "arn": "arn:aws:secretsmanager:us-
west-2:123456789012:secret:MyGreengrassSecret-abcdef"
    }
  ],
  "periodicRefreshIntervalMin" : 60
}
```

`cloudSecrets`

A list of Secrets Manager secrets to deploy to the core device. You can specify labels to define which versions of each secret to deploy. If you don't specify a version, this component deploys the version with the staging label `AWSCURRENT` attached. For more information, see Staging labels in the *AWS Secrets Manager User Guide*.

The secret manager component caches secrets locally. If the secret value changes in Secrets Manager, this component doesn't automatically retrieve the new value. To update the local copy, give the secret a new label and configure this component to retrieve the secret identified by the new label.

Each object contains the following information:

`arn`

The ARN of the secret to deploy. The ARN of the secret can either be a full ARN or a partial ARN. We recommend that you specify a complete ARN rather than a partial ARN. For more information, see Finding a secret from a partial ARN. The following is an example of a full ARN and a partial ARN:

- Full ARN: `arn:aws:secretsmanager:us-east-2:111122223333:secret:SecretName-abcdef`
- Partial ARN: `arn:aws:secretsmanager:us-east-2:111122223333:secret:SecretName`

`labels`

(Optional) A list of labels to identify the versions of the secret to deploy to the core device.

Each label must be a string.

Example: Configuration merge update

```
{
  "cloudSecrets": [
    {
      "arn": "arn:aws:secretsmanager:us-west-2:123456789012:secret:MyGreengrassSecret-abcdef"
    }
  ]
}
```

Local log file

This component uses the same log file as the Greengrass nucleus component.

Linux

```
/greengrass/v2/logs/greengrass.log
```

Windows

```
C:\greengrass\v2\logs\greengrass.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

Linux

```
sudo tail -f /greengrass/v2/logs/greengrass.log
```

Windows (PowerShell)

```
Get-Content C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
```

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.2.10	Bug fixes and improvements • Fixes an issue where a runtime error occurs when you deploy secret manager with the Client device auth component.
2.2.9	Bug fixes and improvements • General bug fixes and improvements.
2.2.8	Updates the component version for the Greengrass nucleus version 2.17.0 release.
2.2.7	Version updated for Greengrass nucleus version 2.16.0 release.
2.2.6	Bug fixes and improvements Fixes an issue where secret manager fails to get a secret due to a slow or unresponsive Trusted Platform Module. Version updated for Greengrass nucleus version 2.15.0 release.
2.2.5	Bug fixes and improvements Fixes an issue where a secret doesn't get fetched from the AWS Cloud if it isn't present in the local cache.

Version	Changes
2.2.4	Bug fixes and improvements Reduces the frequency of writes to the local secret store. Secret manager now writes to the local store only when secrets are updated.
2.2.3	Bug fixes and improvements Fixes an issue where secret manager wipes the locally persisted secrets when the core device is offline and the device's security service (such as an HSM) is unavailable.
2.2.2	Bug fixes and improvements Fixes an issue where secret manager doesn't download the secrets configured with partial arns.
2.2.1	Bug fixes and improvements Supports secret manager on Nucleus versions 2.5.0 and above.
2.2.0	New features Adds support for periodic refresh of the configured secrets through a new component configuration key. Adds support for a new request parameter in the `GetSecretValue` IPC request to refresh the secrets per request
2.1.8	Bug fixes and improvements Fixes an issue where secret manager doesn't accept a partial arn.
2.1.7	Version updated for Greengrass nucleus version 2.12.0 release.
2.1.6	Version updated for Greengrass nucleus version 2.11.0 release.
2.1.5	Version updated for Greengrass nucleus version 2.10.0 release.

Version	Changes
2.1.4	Bug fixes and improvements Fixes an issue where cached secrets were being removed when secret manager is deployed and Greengrass nucleus restarts. Version updated for Greengrass nucleus version 2.9.0 release.
2.1.3	Version updated for Greengrass nucleus version 2.8.0 release.
2.1.2	Version updated for Greengrass nucleus version 2.7.0 release.
2.1.1	Version updated for Greengrass nucleus version 2.6.0 release.
2.1.0	New features • Adds support for hardware security integration. The secret manager component can encrypt and decrypt secrets using a private key that you store in a hardware security module (HSM). For more information, see Hardware security integration. Bug fixes and improvements • Version updated for Greengrass nucleus version 2.5.0 release.
2.0.9	Version updated for Greengrass nucleus version 2.4.0 release.
2.0.8	Version updated for Greengrass nucleus version 2.3.0 release.
2.0.7	Version updated for Greengrass nucleus version 2.2.0 release.
2.0.6	Version updated for Greengrass nucleus version 2.1.0 release.
2.0.5	Improvements • Add support for AWS China Regions and AWS GovCloud (US) Regions.
2.0.4	Initial version.

Secure tunneling

With the `aws.greengrass.SecureTunneling` component, you can establish secure bidirectional communication with a Greengrass core device located behind restricted firewalls.

For example, imagine you have a Greengrass core device behind a firewall that prohibits all incoming connections. Secure tunneling uses MQTT to transfer an access token to the device and then uses WebSockets to make an SSH connection to the device through the firewall. With this AWS IoT managed tunnel, you can open the SSH connection needed for your device. For more information about using AWS IoT secure tunneling to connect to remote devices, see AWS IoT secure tunneling in the *AWS IoT Developer Guide*.

This component subscribes to the AWS IoT Core MQTT message broker on the `$aws/things/greengrass-core-device/tunnels/notify` topic to receive secure tunneling notifications.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Local log file
- Licenses
- Usage
- See also
- Changelog

Versions

This component has the following versions:

- 2.0.x

- 1.1.x
- 1.0.x

Type

This component is a generic component (`aws.greengrass.generic`). The Greengrass nucleus runs the component's lifecycle scripts.

For more information, see Component types.

Operating system

This component can be installed on Linux core devices only.

Architectures:

- Armv71
- Armv8 (AArch64)
- x86_64

Requirements

This component has the following requirements:

2.0.x

- Minimum of 4 MB disk space available for the secure tunneling component. This requirement does not include the Greengrass core software or other components running on the same device.
- Minimum of 3 MB RAM available for the secure tunneling component. This requirement does not include the Greengrass core software or other components running on the same device.
- GNU C Library (glibc) version 2.35 or greater. Versions of the operating system and libraries past their long-term support end of life date are not supported. You should use an operating system and libraries with long-term support.
- The following runtime libraries installed on the Greengrass core device:
  - `libstdc++` version 3.4.29 or greater

  - `libgcc_s` version 3.0 or greater
  - OpenSSL version 3.0.0 or greater
- Open outbound traffic on port 443 on the Greengrass core device.
- Turn on support for the communication service that you want to use to communicate with the Greengrass core device. For example, to open an SSH connection to the device, you must turn on SSH on that device.

1.0.x - 1.1.x

- Minimum of 32 MB disk space available for the secure tunneling component. This requirement does not include the Greengrass core software or other components running on the same device.
- Minimum of 16 MB RAM available for the secure tunneling component. This requirement does not include the Greengrass core software or other components running on the same device. For more information, see Control memory allocation with JVM options.
- GNU C Library (glibc) version 2.25 or greater with a Linux kernel of 3.2 or greater are required for the secure tunneling component version 1.0.12 and greater. Versions of the operating system and libraries past their long-term support end of life date are not supported. You should use an operating system and libraries with long-term support.
- Both the operating system and the Java runtime must be installed as 64 bit.
- Python 3.5 or later installed on the Greengrass core device and added to the PATH environment variable.
- `libcrypto.so.1.1` installed on the Greengrass core device and added to the PATH environment variable.
- Open outbound traffic on port 443 on the Greengrass core device.
- Turn on support for the communication service that you want to use to communicate with the Greengrass core device. For example, to open an SSH connection to the device, you must turn on SSH on that device.

Endpoints and ports

This component must be able to perform outbound requests to the following endpoints and ports, in addition to endpoints and ports required for basic operation. For more information, see Allow device traffic through a proxy or firewall.

Endpoint	Port	Required	Description
`data.tunneling.iot. region.amazonaws.com`	443	Yes	Establish secure tunnels.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

2.0.0

Version 2.0.0 of this component supports both Greengrass nucleus and Greengrass nucleus lite. The following table lists the dependencies for version 2.0.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus OR Greengrass nucleus lite	>=2.0.0 <3.0.0	Soft

1.0.19 – 1.1.3

The following table lists the dependencies for versions 1.0.19 through 1.1.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <3.0.0	Soft

1.0.18

The following table lists the dependencies for version 1.0.18 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.13.0	Soft

1.0.16 – 1.0.17

The following table lists the dependencies for versions 1.0.16 to 1.0.17 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.12.0	Soft

1.0.14 – 1.0.15

The following table lists the dependencies for versions 1.0.14 to 1.0.15 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.11.0	Soft

1.0.11 – 1.0.13

The following table lists the dependencies for versions 1.0.11 – 1.0.13 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.10.0	Soft

1.0.10

The following table lists the dependencies for version 1.0.10 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.9.0	Soft

1.0.9

The following table lists the dependencies for version 1.0.9 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.8.0	Soft

1.0.8

The following table lists the dependencies for version 1.0.8 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.7.0	Soft

1.0.5 - 1.0.7

The following table lists the dependencies for versions 1.0.5 through 1.0.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.6.0	Soft

1.0.4

The following table lists the dependencies for version 1.0.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.5.0	Soft

1.0.3

The following table lists the dependencies for version 1.0.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.4.0	Soft

1.0.2

The following table lists the dependencies for version 1.0.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.3.0	Soft

1.0.1

The following table lists the dependencies for version 1.0.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.2.0	Soft

1.0.0

The following table lists the dependencies for version 1.0.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.3 <2.1.0	Soft

For more information about component dependencies, see the component recipe reference.

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

2.0.x

`maxConcurrentTunnels`

(Optional) Maximum number of concurrent tunnels allowed.

> **Note**
>
> The value cannot be more than 20.

Default: `20`

`tunnelTimeoutSeconds`

(Optional) Tunnel timeout duration in seconds.

Default: `43200` (12 hours)

`accessControl`

(Optional) The object that contains the authorization policy that allows the component to subscribe to the secure tunneling notifications topic.

> **Note**
>
> Do not modify this configuration parameter if your deployment targets a thing group. If your deployment targets an individual core device, and you want to restrict its subscription to the device's topic, specify the core device's thing name. In the `resources` value in the device's authorization policy, replace the MQTT topic wildcard with the device's thing name.

```
{
  "aws.greengrass.ipc.mqttproxy": {
    "aws.greengrass.SecureTunneling:mqttproxy:1": {
      "policyDescription": "Access to tunnel notification pubsub topic",
```

```
      "operations": [
        "aws.greengrass#SubscribeToIoTCore"
      ],
      "resources": [
        "$aws/things/+/tunnels/notify"
      ]
    }
  }
}
```

Example: Configuration merge update

The following example configuration specifies the maximum concurrent tunnels and timeout settings.

```
{
  "maxConcurrentTunnels": 20,
  "tunnelTimeoutSeconds": 43200,
  "accessControl": {
    "aws.greengrass.ipc.mqttproxy": {
      "aws.greengrass.SecureTunneling:mqttproxy:1": {
        "policyDescription": "Access to tunnel notification pubsub topic",
        "operations": [
          "aws.greengrass#SubscribeToIoTCore"
        ],
        "resources": [
          "$aws/things/MyGreengrassCore/tunnels/notify"
        ]
      }
    }
  }
}
```

1.0.x - 1.1.x

`OS_DIST_INFO`

(Optional) The operating system of your core device. By default, the component attempts to identify automatically the operating system running on your core device. If the component fails to start with the default value, use this value to specify the operating system. For a list of supported operating systems for this component, see Device requirements.

This value can be one of the following: `auto`, `ubuntu`, `amzn2`, `raspberrypi`.

Default: `auto`

`accessControl`

(Optional) The object that contains the authorization policy that allows the component to subscribe to the secure tunneling notifications topic.

> **Note**
>
> Do not modify this configuration parameter if your deployment targets a thing group. If your deployment targets an individual core device, and you want to restrict its subscription to the device's topic, specify the core device's thing name. In the `resources` value in the device's authorization policy, replace the MQTT topic wildcard with the device's thing name.

```
{
  "aws.greengrass.ipc.mqttproxy": {
    "aws.iot.SecureTunneling:mqttproxy:1": {
      "policyDescription": "Access to tunnel notification pubsub topic",
      "operations": [
        "aws.greengrass#SubscribeToIoTCore"
      ],
      "resources": [
        "$aws/things/+/tunnels/notify"
      ]
    }
  }
}
```

Example: Configuration merge update

The following example configuration specifies to allow this component to open secure tunnels on a core device named **`MyGreengrassCore`** that runs Ubuntu.

```
{
  "OS_DIST_INFO": "ubuntu",
  "accessControl": {
```

```
    "aws.greengrass.ipc.mqttproxy": {
      "aws.iot.SecureTunneling:mqttproxy:1": {
        "policyDescription": "Access to tunnel notification pubsub topic",
        "operations": [
          "aws.greengrass#SubscribeToIoTCore"
        ],
        "resources": [
          "$aws/things/MyGreengrassCore/tunnels/notify"
        ]
      }
    }
  }
}
```

Local log file

Greengrass nucleus

This component uses the following log file.

```
/greengrass/v2/logs/aws.greengrass.SecureTunneling.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* with the path to the AWS IoT Greengrass root folder.

  ```
  sudo tail -f /greengrass/v2/logs/aws.greengrass.SecureTunneling.log
  ```

Greengrass nucleus lite

Run the following command on the core device to view this component's logs.

```
journalctl -xeau ggl.aws.greengrass.SecureTunneling.service
```

Run the following command on the core device to view this component's logs in real time.

```
journalctl -fau ggl.aws.greengrass.SecureTunneling.service
```

Licenses

This component includes the following third-party software/licensing:

2.0.x

- AWS IoT Secure Tunneling Local Proxy/Apache License 2.0
- AWS Greengrass Component SDK/Apache License 2.0
- Boost C++ Libraries/Boost Software License 1.0
- Protocol Buffers/BSD 3-Clause License

1.0.x - 1.1.x

- AWS IoT Device Client/Apache License 2.0
- AWS IoT Device SDK for Java/Apache License 2.0
- gson/Apache License 2.0
- log4j/Apache License 2.0
- slf4j/Apache License 2.0

Usage

To use the secure tunneling component on your device, do the following:

1. Deploy the secure tunneling component to your device.
2. Open the AWS IoT console. From the left menu, choose **Remote actions**, and then choose **Secure tunnels**.
3. Create a tunnel to your Greengrass device.
4. Download the source access token.
5. Use the local proxy with the source access token to connect to your destination. For more information, see How to use the local proxy in the *AWS IoT Developer Guide*.

See also

- AWS IoT secure tunneling in the *AWS IoT Developer Guide*
- How to use the local proxy in the *AWS IoT Developer Guide*

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.0.1	Bug fixes and improvements • Updates the bundled local proxy binary to version 3.3.0. • Fixes concurrency and object-lifetime bugs that could affect tunnel data integrity. • Requires TLS 1.2 or later and no longer allows SSLv2, SSLv3, TLS 1.0, or TLS 1.1.
2.0.0	New features • Replaces the Java wrapper with a C wrapper. • Replaces AWS IoT Device Client with AWS IoT Securetunneling Localproxy. • Reduces the binary size from approximately 36 MB to approximately 4 MB. • Reduces the memory footprint from approximately 100 MB to approximately 2 MB.
1.1.3	Bug fixes and improvements • Upgrades the underlying AWS IoT Device Client invoked by the component from version 1.10.0 to version 1.10.1. • Fixes the GNU C Library (glibc) compatibility issues in secure tunneling component version 1.1.2.
1.1.2	**Warning** This version is no longer available. The improvements in this version are available in later versions of this component.

Version	Changes
	Bug fixes and improvements • Upgrades the underlying AWS IoT Device Client invoked by the component from version 1.9.0 to version 1.10.0. • Fixes the payload transfer issue which prevents users from forwardin g large files from Greengrass V2 core devices to the source device through the secure tunnel.
1.1.1	Bug fixes and improvements • Adds a configuration to support Greengrass nucleus lite.
1.1.0	New features • Adds recipe supports for Greengrass nucleus lite.
1.0.19	Bug fixes and improvements • Upgrades the underlying AWS IoT Device Client invoked by the component from version 1.8.0 to version 1.9.0. • Increases the concurrent tunnel limit to 20 tunnels on a component level. • Increases the default AWS IoT Greengrass Core IPC timeout from 3 seconds to 10 seconds. **Warning** If you are using the secure tunneling local proxy as the tunnel source client, do not update your component to this version until you have also upgraded the local proxy to version 3.1.1 or later.
1.0.18	Version updated for Greengrass nucleus version 2.12.0 release.

Version	Changes
1.0.17	Bug fixes and improvements • Fixes the thread cleanup issue which was blocking users from creating tunnels. This component will now cleanup a thread either once it receives the CloseTunnel signal or if the tunnel is expired after 12 hours.
1.0.16	Version updated for Greengrass nucleus version 2.11.0 release.
1.0.15	Bug fixes and improvements • Fixes a startup issue for users that do not have a home directory on the device. The secure tunneling component now starts without creating a directory for shadow documents.
1.0.14	Version updated for Greengrass nucleus version 2.10.0 release.
1.0.13	Bug fixes and improvements • Fixes an issue where an orphan client process prevents more than one tunnel from targeting the device.
1.0.12	Bug fixes and improvements • Adds support for x86_64 (AMD64) and ARMv8 (Aarch64) when running on Raspberry Pi OS.
1.0.11	Version updated for Greengrass nucleus version 2.9.0 release.
1.0.10	Version updated for Greengrass nucleus version 2.8.0 release.
1.0.9	Version updated for Greengrass nucleus version 2.7.0 release.
1.0.8	Version updated for Greengrass nucleus version 2.6.0 release.
1.0.7	Bug fixes and improvements • Fixes an issue where the component disconnects when you transfer large files over SCP.
1.0.6	This version contains bug fixes.

Version	Changes
1.0.5	Version updated for Greengrass nucleus version 2.5.0 release.
1.0.4	Version updated for Greengrass nucleus version 2.4.0 release.
1.0.3	Version updated for Greengrass nucleus version 2.3.0 release.
1.0.2	Version updated for Greengrass nucleus version 2.2.0 release.
1.0.1	Version updated for Greengrass nucleus version 2.1.0 release.
1.0.0	Initial version.

Shadow manager

The shadow manager component (`aws.greengrass.ShadowManager`) enables the local shadow service on your core device. The local shadow service allows components to use interprocess communication to interact with local shadows. The shadow manager component manages the storage of local shadow documents, and also handles synchronization of local shadow states with the AWS IoT Device Shadow service.

For more information about how Greengrass core devices can interact with shadows, see Interact with device shadows.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Local log file
- Changelog

Versions

This component has the following versions:

- 2.3.x
- 2.2.x
- 2.1.x
- 2.0.x

Type

This component is a plugin component (`aws.greengrass.plugin`). The Greengrass nucleus runs this component in the same Java Virtual Machine (JVM) as the nucleus. The nucleus restarts when you change this component's version on the core device.

This component uses the same log file as the Greengrass nucleus. For more information, see Monitor AWS IoT Greengrass logs.

For more information, see Component types.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

Requirements

This component has the following requirements:

- (Optional) To sync shadows to the AWS IoT Device Shadow service, the Greengrass core device's AWS IoT policy must allow the following AWS IoT Core shadow policy actions:
  - `iot:GetThingShadow`
  - `iot:UpdateThingShadow`
  - `iot:DeleteThingShadow`

For more information about these AWS IoT Core policies, see AWS IoT Core policy actions in the *AWS IoT Developer Guide*.

For more information about the minimal AWS IoT policy, see Minimal AWS IoT policy for AWS IoT Greengrass V2 core devices

- The shadow manager component is supported to run in a VPC.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

2.3.15

The following table lists the dependencies for version 2.3.15 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.19.0	Soft

2.3.14

The following table lists the dependencies for version 2.3.14 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.18.0	Soft

2.3.13

The following table lists the dependencies for version 2.3.13 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.17.0	Soft

2.3.12

The following table lists the dependencies for version 2.3.12 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.17.0	Soft

2.3.11

The following table lists the dependencies for version 2.3.11 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.16.0	Soft

2.3.10

The following table lists the dependencies for version 2.3.10 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.15.0	Soft

2.3.9

The following table lists the dependencies for version 2.3.9 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.14.0	Soft

2.3.5 – 2.3.8

The following table lists the dependencies for versions 2.3.5 through 2.3.8 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.13.0	Soft

2.3.3 and 2.3.4

The following table lists the dependencies for versions 2.3.3 and 2.3.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.12.0	Soft

2.3.2

The following table lists the dependencies for version 2.3.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.11.0	Soft

2.3.0 and 2.3.1

The following table lists the dependencies for versions 2.3.0 and 2.3.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.5.0 <2.10.0	Soft

2.2.3 and 2.2.4

The following table lists the dependencies for versions 2.2.3 and 2.2.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.2.0 <3.0.0	Soft

2.2.2

The following table lists the dependencies for version 2.2.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.2.0 <2.9.0	Soft

2.2.1

The following table lists the dependencies for version 2.2.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.2.0 <2.8.0	Soft

2.1.1 and 2.2.0

The following table lists the dependencies for versions 2.1.1 and 2.2.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.2.0 <2.7.0	Soft

2.0.5 - 2.1.0

The following table lists the dependencies for versions 2.0.5 through 2.1.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.2.0 <2.6.0	Soft

2.0.3 and 2.0.4

The following table lists the dependencies for versions 2.0.3 and 2.0.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.2.0 <2.5.0	Soft

2.0.1 and 2.0.2

The following table lists the dependencies for versions 2.0.1 and 2.0.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.2.0 <2.4.0	Soft

2.0.0

The following table lists the dependencies for version 2.0.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.2.0 <2.3.0	Soft

For more information about component dependencies, see the component recipe reference.

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

2.3.x

`strategy`

(Optional) The strategy that this component uses to sync shadows between AWS IoT Core and the core device.

This object contains the following information.

`type`

(Optional) The type of strategy that this component uses to sync shadows between AWS IoT Core and the core device. Choose from the following options:

- `realTime` – Sync shadows with AWS IoT Core each time a shadow update occurs.
- `periodic` – Sync shadows with AWS IoT Core on a regular interval that you specify with the `delay` configuration parameter.

Default: `realTime`

`delay`

(Optional) The interval in seconds where this component syncs shadows with AWS IoT Core, when you specify the `periodic` sync strategy.

> **Note**
>
> This parameter is required if you specify the `periodic` sync strategy.

`synchronize`

(Optional) The synchronization settings that determine how shadows are synced with the AWS Cloud.

> **Note**
>
> You must create a configuration update with this property to sync shadows with the AWS Cloud.

This object contains the following information.

`coreThing`

(Optional) The core device shadows to sync. This object contains the following information.

`classic`

(Optional) By default, the shadow manager syncs the local state of the classic shadow for your core device with the AWS Cloud. If you don't want to sync the classic device shadow, set this to `false`.

Default: `true`

`namedShadows`

(Optional) The list of named core device shadows to sync. You must specify the exact names of the shadows.

> **Warning**
>
> The AWS IoT Greengrass service uses the `AWSManagedGreengrassV2Deployment` named shadow to manage deployments that target individual core devices. This named shadow is reserved for use by the AWS IoT Greengrass service. Do not update or delete this named shadow.

`shadowDocumentsMap`

(Optional) The additional device shadows to sync. Using this configuration parameter makes it easier to specify shadow documents. We recommend that you use this parameter instead of the `shadowDocuments` object.

> **Note**
>
> If you specify a `shadowDocumentsMap` object, you must not specify a `shadowDocuments` object.

Each object contains the following information:

`thingName`

The shadow configuration for the *`thingName`* for this shadow configuration.

`classic`

(Optional) If you don't want to sync the classic device shadow for the `thingName` device, set this to `false`.

`namedShadows`

The list of named shadows that you want to sync. You must specify the exact names of the shadows.

`shadowDocuments`

(Optional) The list of additional device shadows to sync. We recommend that you use the `shadowDocumentsMap` parameter instead.

> **Note**
>
> If you specify a `shadowDocuments` object, you must not specify a `shadowDocumentsMap` object.

Each object in this list contains the following information.

`thingName`

The thing name of the device for which to sync shadows.

`classic`

(Optional) If you don't want to sync the classic device shadow for the `thingName` device, set this to `false`.

Default: `true`

`namedShadows`

(Optional) The list of named device shadows that you want to sync. You must specify the exact names of the shadows.

`direction`

(Optional) The direction to sync shadows between the local shadow service and the AWS Cloud. You can configure this option to reduce bandwidth and connections to the AWS Cloud. Choose from the following options:

- `betweenDeviceAndCloud` – Synchronize shadows between the local shadow service and the AWS Cloud.
- `deviceToCloud` – Send shadow updates from the local shadow service to the AWS Cloud, and ignore shadow updates from the AWS Cloud.
- `cloudToDevice` – Receive shadow updates from the AWS Cloud, and don't send shadow updates from the local shadow service to the AWS Cloud.

Default: `BETWEEN_DEVICE_AND_CLOUD`

`rateLimits`

(Optional) The settings that determine the rate limits for shadow service requests.

This object contains the following information.

`maxOutboundSyncUpdatesPerSecond`

(Optional) The maximum number of sync requests per second that the device transmits.

Default: 100 requests/second

`maxTotalLocalRequestsRate`

(Optional) The maximum number of local IPC requests per second that are sent to the core device.

Default: 200 requests/second

`maxLocalRequestsPerSecondPerThing`

(Optional) The maximum number of local IPC requests per second that are sent for each connected IoT thing.

Default: 20 requests/second for each thing

> **Note**
>
> These rate limits parameters define the maximum number of requests per second for the local shadow service. The maximum number of requests per second for the AWS IoT Device Shadow service depends on your AWS Region. For more information, see the limits for the AWS IoT Device Shadow Service API in the *Amazon Web Services General Reference*.

`shadowDocumentSizeLimitBytes`

(Optional) The maximum allowed size of each JSON state document for local shadows.

If you increase this value, you must also increase the resource limit for the JSON state document for cloud shadows. For more information, see the limits for the AWS IoT Device Shadow Service API in the *Amazon Web Services General Reference*.

Default: 8192 bytes

Maximum: 30720 bytes

Example: Configuration merge update

The following example shows a sample configuration merge update with all available configuration parameters for the shadow manager component.

```
{
   "strategy":{
      "type":"periodic",
      "delay":300
   },
   "synchronize":{
      "shadowDocumentsMap":{
         "MyDevice1":{
            "classic":false,
            "namedShadows":[
               "MyShadowA",
               "MyShadowB"
            ]
         },
         "MyDevice2":{
            "classic":true,
            "namedShadows":[]
         }
      },
      "direction":"betweenDeviceAndCloud"
   },
   "rateLimits":{
      "maxOutboundSyncUpdatesPerSecond":100,
      "maxTotalLocalRequestsRate":200,
      "maxLocalRequestsPerSecondPerThing":20
   },
```

```
    "shadowDocumentSizeLimitBytes":8192
}
```

2.2.x

`strategy`

(Optional) The strategy that this component uses to sync shadows between AWS IoT Core and the core device.

This object contains the following information.

`type`

(Optional) The type of strategy that this component uses to sync shadows between AWS IoT Core and the core device. Choose from the following options:

- `realTime` – Sync shadows with AWS IoT Core each time a shadow update occurs.
- `periodic` – Sync shadows with AWS IoT Core on a regular interval that you specify with the `delay` configuration parameter.

Default: `realTime`

`delay`

(Optional) The interval in seconds where this component syncs shadows with AWS IoT Core, when you specify the `periodic` sync strategy.

> **Note**
>
> This parameter is required if you specify the `periodic` sync strategy.

`synchronize`

(Optional) The synchronization settings that determine how shadows are synced with the AWS Cloud.

> **Note**
>
> You must create a configuration update with this property to sync shadows with the AWS Cloud.

This object contains the following information.

`coreThing`

(Optional) The core device shadows to sync. This object contains the following information.

`classic`

(Optional) By default, the shadow manager syncs the local state of the classic shadow for your core device with the AWS Cloud. If you don't want to sync the classic device shadow, set this to `false`.

Default: `true`

`namedShadows`

(Optional) The list of named core device shadows to sync. You must specify the exact names of the shadows.

> **Warning**
>
> The AWS IoT Greengrass service uses the `AWSManagedGreengrassV2Deployment` named shadow to manage deployments that target individual core devices. This named shadow is reserved for use by the AWS IoT Greengrass service. Do not update or delete this named shadow.

`shadowDocumentsMap`

(Optional) The additional device shadows to sync. Using this configuration parameter makes it easier to specify shadow documents. We recommend that you use this parameter instead of the `shadowDocuments` object.

> **Note**
>
> If you specify a `shadowDocumentsMap` object, you must not specify a `shadowDocuments` object.

Each object contains the following information:

`thingName`

The shadow configuration for the *`thingName`* for this shadow configuration.

`classic`

(Optional) If you don't want to sync the classic device shadow for the `thingName` device, set this to `false`.

`namedShadows`

The list of named shadows that you want to sync. You must specify the exact names of the shadows.

`shadowDocuments`

(Optional) The list of additional device shadows to sync. We recommend that you use the `shadowDocumentsMap` parameter instead.

> **Note**
>
> If you specify a `shadowDocuments` object, you must not specify a `shadowDocumentsMap` object.

Each object in this list contains the following information.

`thingName`

The thing name of the device for which to sync shadows.

`classic`

(Optional) If you don't want to sync the classic device shadow for the `thingName` device, set this to `false`.

Default: `true`

`namedShadows`

(Optional) The list of named device shadows that you want to sync. You must specify the exact names of the shadows.

`direction`

(Optional) The direction to sync shadows between the local shadow service and the AWS Cloud. You can configure this option to reduce bandwidth and connections to the AWS Cloud. Choose from the following options:

- `betweenDeviceAndCloud` – Synchronize shadows between the local shadow service and the AWS Cloud.
- `deviceToCloud` – Send shadow updates from the local shadow service to the AWS Cloud, and ignore shadow updates from the AWS Cloud.
- `cloudToDevice` – Receive shadow updates from the AWS Cloud, and don't send shadow updates from the local shadow service to the AWS Cloud.

Default: `BETWEEN_DEVICE_AND_CLOUD`

`rateLimits`

(Optional) The settings that determine the rate limits for shadow service requests.

This object contains the following information.

`maxOutboundSyncUpdatesPerSecond`

(Optional) The maximum number of sync requests per second that the device transmits.

Default: 100 requests/second

`maxTotalLocalRequestsRate`

(Optional) The maximum number of local IPC requests per second that are sent to the core device.

Default: 200 requests/second

`maxLocalRequestsPerSecondPerThing`

(Optional) The maximum number of local IPC requests per second that are sent for each connected IoT thing.

Default: 20 requests/second for each thing

> **Note**
>
> These rate limits parameters define the maximum number of requests per second for the local shadow service. The maximum number of requests per second for the

AWS IoT Device Shadow service depends on your AWS Region. For more information, see the limits for the AWS IoT Device Shadow Service API in the *Amazon Web Services General Reference*.

`shadowDocumentSizeLimitBytes`

(Optional) The maximum allowed size of each JSON state document for local shadows.

If you increase this value, you must also increase the resource limit for the JSON state document for cloud shadows. For more information, see the limits for the AWS IoT Device Shadow Service API in the *Amazon Web Services General Reference*.

Default: 8192 bytes

Maximum: 30720 bytes

Example: Configuration merge update

The following example shows a sample configuration merge update with all available configuration parameters for the shadow manager component.

```
{
   "strategy":{
      "type":"periodic",
      "delay":300
   },
   "synchronize":{
      "shadowDocumentsMap":{
         "MyDevice1":{
            "classic":false,
            "namedShadows":[
               "MyShadowA",
               "MyShadowB"
            ]
         },
         "MyDevice2":{
            "classic":true,
            "namedShadows":[]
         }
      },
      "direction":"betweenDeviceAndCloud"
```

```
    },
    "rateLimits":{
       "maxOutboundSyncUpdatesPerSecond":100,
       "maxTotalLocalRequestsRate":200,
       "maxLocalRequestsPerSecondPerThing":20
    },
    "shadowDocumentSizeLimitBytes":8192
}
```

2.1.x

`strategy`

(Optional) The strategy that this component uses to sync shadows between AWS IoT Core and the core device.

This object contains the following information.

`type`

(Optional) The type of strategy that this component uses to sync shadows between AWS IoT Core and the core device. Choose from the following options:

- `realTime` – Sync shadows with AWS IoT Core each time a shadow update occurs.
- `periodic` – Sync shadows with AWS IoT Core on a regular interval that you specify with the `delay` configuration parameter.

Default: `realTime`

`delay`

(Optional) The interval in seconds where this component syncs shadows with AWS IoT Core, when you specify the `periodic` sync strategy.

> **Note**
>
> This parameter is required if you specify the `periodic` sync strategy.

`synchronize`

(Optional) The synchronization settings that determine how shadows are synced with the AWS Cloud.

> **Note**
>
> You must create a configuration update with this property to sync shadows with the AWS Cloud.

This object contains the following information.

`coreThing`

(Optional) The core device shadows to sync. This object contains the following information.

`classic`

(Optional) By default, the shadow manager syncs the local state of the classic shadow for your core device with the AWS Cloud. If you don't want to sync the classic device shadow, set this to `false`.

Default: `true`

`namedShadows`

(Optional) The list of named core device shadows to sync. You must specify the exact names of the shadows.

> **Warning**
>
> The AWS IoT Greengrass service uses the `AWSManagedGreengrassV2Deployment` named shadow to manage deployments that target individual core devices. This named shadow is reserved for use by the AWS IoT Greengrass service. Do not update or delete this named shadow.

`shadowDocumentsMap`

(Optional) The additional device shadows to sync. Using this configuration parameter makes it easier to specify shadow documents. We recommend that you use this parameter instead of the `shadowDocuments` object.

> **Note**
>
> If you specify a `shadowDocumentsMap` object, you must not specify a `shadowDocuments` object.

Each object contains the following information:

`thingName`

The shadow configuration for the *`thingName`* for this shadow configuration.

`classic`

(Optional) If you don't want to sync the classic device shadow for the `thingName` device, set this to `false`.

`namedShadows`

The list of named shadows that you want to sync. You must specify the exact names of the shadows.

`shadowDocuments`

(Optional) The list of additional device shadows to sync. We recommend that you use the `shadowDocumentsMap` parameter instead.

> **Note**
>
> If you specify a `shadowDocuments` object, you must not specify a `shadowDocumentsMap` object.

Each object in this list contains the following information.

`thingName`

The thing name of the device for which to sync shadows.

`classic`

(Optional) If you don't want to sync the classic device shadow for the `thingName` device, set this to `false`.

Default: `true`

`namedShadows`

(Optional) The list of named device shadows that you want to sync. You must specify the exact names of the shadows.

`rateLimits`

(Optional) The settings that determine the rate limits for shadow service requests.

This object contains the following information.

`maxOutboundSyncUpdatesPerSecond`

(Optional) The maximum number of sync requests per second that the device transmits.

Default: 100 requests/second

`maxTotalLocalRequestsRate`

(Optional) The maximum number of local IPC requests per second that are sent to the core device.

Default: 200 requests/second

`maxLocalRequestsPerSecondPerThing`

(Optional) The maximum number of local IPC requests per second that are sent for each connected IoT thing.

Default: 20 requests/second for each thing

> **Note**
>
> These rate limits parameters define the maximum number of requests per second for the local shadow service. The maximum number of requests per second for the AWS IoT Device Shadow service depends on your AWS Region. For more information, see the limits for the AWS IoT Device Shadow Service API in the *Amazon Web Services General Reference*.

`shadowDocumentSizeLimitBytes`

(Optional) The maximum allowed size of each JSON state document for local shadows.

If you increase this value, you must also increase the resource limit for the JSON state document for cloud shadows. For more information, see the limits for the AWS IoT Device Shadow Service API in the *Amazon Web Services General Reference*.

Default: 8192 bytes

Maximum: 30720 bytes

Example: Configuration merge update

The following example shows a sample configuration merge update with all available configuration parameters for the shadow manager component.

```
{
   "strategy":{
      "type":"periodic",
      "delay":300
   },
   "synchronize":{
      "shadowDocumentsMap":{
         "MyDevice1":{
            "classic":false,
            "namedShadows":[
               "MyShadowA",
               "MyShadowB"
            ]
         },
         "MyDevice2":{
            "classic":true,
            "namedShadows":[]
         }
      },
      "direction":"betweenDeviceAndCloud"
   },
   "rateLimits":{
      "maxOutboundSyncUpdatesPerSecond":100,
      "maxTotalLocalRequestsRate":200,
      "maxLocalRequestsPerSecondPerThing":20
   },
   "shadowDocumentSizeLimitBytes":8192
}
```

2.0.x

`synchronize`

(Optional) The synchronization settings that determine how shadows are synced with the AWS Cloud.

> **Note**
>
> You must create a configuration update with this property to sync shadows with the AWS Cloud.

This object contains the following information.

`coreThing`

(Optional) The core device shadows to sync. This object contains the following information.

`classic`

(Optional) By default, the shadow manager syncs the local state of the classic shadow for your core device with the AWS Cloud. If you don't want to sync the classic device shadow, set this to `false`.

Default: `true`

`namedShadows`

(Optional) The list of named core device shadows to sync. You must specify the exact names of the shadows.

> **Warning**
>
> The AWS IoT Greengrass service uses the `AWSManagedGreengrassV2Deployment` named shadow to manage deployments that target individual core devices. This named shadow is reserved for use by the AWS IoT Greengrass service. Do not update or delete this named shadow.

`shadowDocumentsMap`

(Optional) The additional device shadows to sync. Using this configuration parameter makes it easier to specify shadow documents. We recommend that you use this parameter instead of the `shadowDocuments` object.

> **Note**
>
> If you specify a `shadowDocumentsMap` object, you must not specify a `shadowDocuments` object.

Each object contains the following information:

`thingName`

The shadow configuration for the *`thingName`* for this shadow configuration.

`classic`

(Optional) If you don't want to sync the classic device shadow for the `thingName` device, set this to `false`.

`namedShadows`

The list of named shadows that you want to sync. You must specify the exact names of the shadows.

`shadowDocuments`

(Optional) The list of additional device shadows to sync. We recommend that you use the `shadowDocumentsMap` parameter instead.

> **Note**
>
> If you specify a `shadowDocuments` object, you must not specify a `shadowDocumentsMap` object.

Each object in this list contains the following information.

`thingName`

The thing name of the device for which to sync shadows.

`classic`

(Optional) If you don't want to sync the classic device shadow for the `thingName` device, set this to `false`.

Default: `true`

`namedShadows`

(Optional) The list of named device shadows that you want to sync. You must specify the exact names of the shadows.

`rateLimits`

(Optional) The settings that determine the rate limits for shadow service requests.

This object contains the following information.

`maxOutboundSyncUpdatesPerSecond`

(Optional) The maximum number of sync requests per second that the device transmits.

Default: 100 requests/second

`maxTotalLocalRequestsRate`

(Optional) The maximum number of local IPC requests per second that are sent to the core device.

Default: 200 requests/second

`maxLocalRequestsPerSecondPerThing`

(Optional) The maximum number of local IPC requests per second that are sent for each connected IoT thing.

Default: 20 requests/second for each thing

> **Note**
>
> These rate limits parameters define the maximum number of requests per second for the local shadow service. The maximum number of requests per second for the AWS IoT Device Shadow service depends on your AWS Region. For more information, see the limits for the AWS IoT Device Shadow Service API in the *Amazon Web Services General Reference*.

`shadowDocumentSizeLimitBytes`

(Optional) The maximum allowed size of each JSON state document for local shadows.

If you increase this value, you must also increase the resource limit for the JSON state document for cloud shadows. For more information, see the limits for the AWS IoT Device Shadow Service API in the *Amazon Web Services General Reference*.

Default: 8192 bytes

Maximum: 30720 bytes

Example: Configuration merge update

The following example shows a sample configuration merge update with all available configuration parameters for the shadow manager component.

```
{
  "synchronize": {
    "coreThing": {
      "classic": true,
      "namedShadows": [
        "MyCoreShadowA",
        "MyCoreShadowB"
      ]
    },
    "shadowDocuments": [
      {
        "thingName": "MyDevice1",
        "classic": false,
        "namedShadows": [
          "MyShadowA",
          "MyShadowB"
        ]
      },
      {
        "thingName": "MyDevice2",
        "classic": true,
        "namedShadows": []
      }
    ]
  },
```

```
  "rateLimits": {
    "maxOutboundSyncUpdatesPerSecond": 100,
    "maxTotalLocalRequestsRate": 200,
    "maxLocalRequestsPerSecondPerThing": 20
  },
  "shadowDocumentSizeLimitBytes": 8192
}
```

Local log file

This component uses the same log file as the Greengrass nucleus component.

Linux

```
/greengrass/v2/logs/greengrass.log
```

Windows

```
C:\greengrass\v2\logs\greengrass.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
  ```

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.3.15	Bug fixes and improvements • Fixes an issue where the shadow manager could stop working after a power outage corrupted its local database. • Validates the integrity of the local database on startup and recreates the database when it detects corruption.
2.3.14	Updates the component version for the Greengrass nucleus version 2.17.0 release.
2.3.13	Bug fixes and improvements • Fixes an issue where local shadow updates would intermittently fail to sync to the cloud even when sync direction configuration allows it. • Minor bug fixes and improvements.
2.3.12	Bug fixes and improvements • Fixes an issue where deployments were blocked when more than 1024 device shadows were configured.
2.3.11	Version updated for Greengrass nucleus version 2.15.0 release.
2.3.10	Version updated for Greengrass nucleus version 2.14.0 release.
2.3.9	Version updated for Greengrass nucleus version 2.13.0 release.
2.3.8	Bug fixes and improvements • Fixes an issue where shadow manager creates a deadlock situation during the MQTT client connection.
2.3.7	Bug fixes and improvements • Fixes an issue where shadow manager periodically logs a `NullPointerException` error during a shadow manager sync.

Version	Changes
2.3.6	Bug fixes and improvements • Fixes an issue where shadow properties that are deleted through AWS Cloud updates while the device is offline continue to exist in the local shadow after regaining connectivity.
2.3.5	Version updated for Greengrass nucleus version 2.12.0 release.
2.3.4	Bug fixes and improvements • Adds support for null and empty shadow state documents.
2.3.3	Version updated for Greengrass nucleus version 2.11.0 release.
2.3.2	Bug fixes and improvements • Fixes an issue where shadow manager enters the `BROKEN` state when the local shadow database is corrupted. • Version updated for Greengrass nucleus version 2.10.0 release.
2.3.1	Bug fixes and improvements • Fixes a condition that may prevent cloud shadow updates from syncing. • Fixes an issue where changes to named shadow sync configuration applies to only one named shadow.
2.3.0	Bug fixes and improvements • Fixes an issue that might prevent shadows from syncing when the Greengrass device private key is stored in a hardware security module.
2.2.4	Bug fixes and improvements • Fixes an issue where the validation of the shadow's size wasn't consistent with the cloud when updating the local shadow document. • Fixes an issue where the shadow manager stops listening to configura tion updates if a deployment performs a `RESET` on the configuration nodes.
2.2.3	Version updated for Greengrass nucleus version 2.9.0 release.

Version	Changes
2.2.2	Version updated for Greengrass nucleus version 2.8.0 release.
2.2.1	Version updated for Greengrass nucleus version 2.7.0 release.
2.2.0	New features • Adds support for the local shadow service over the local publish/subscribe interface. You can now communicate with the local publish/subscribe message broker on shadow MQTT topics to get, update, and delete shadows on the core device. This feature enables you to connect client devices to the local shadow service by using the MQTT bridge to relay messages on shadow topics between client devices and the local publish/subscribe interface. This feature requires v2.6.0 or later of the Greengrass nucleus component. To connect client devices to the local shadow service, you must also use v2.2.0 or later of the MQTT bridge component. • Adds the `direction` option that you can configure to customize the direction to sync shadows between the local shadow service and the AWS Cloud. You can configure this option to reduce bandwidth and connections to the AWS Cloud.
2.1.1	Bug fixes and improvements • Fixes an issue where the maximum depth in the `desired` and `reported` sections of the JSON device shadow state document was 4 levels instead of 5 levels. • Version updated for Greengrass nucleus version 2.6.0 release.
2.1.0	New features • Adds support for periodic shadow synchronization intervals, so you can configure the core device to reduce bandwidth usage and charges.
2.0.6	This version contains bug fixes and improvements.
2.0.5	Version updated for Greengrass nucleus version 2.5.0 release.

Version	Changes
2.0.4	Bug fixes and improvements • Fixes an issue that caused shadow manager to delete newly created versions of any shadow that was previously deleted. • Updates the `DeleteThingShadow` IPC operation to increment the shadow version when called.
2.0.3	Version updated for Greengrass nucleus version 2.4.0 release.
2.0.2	Bug fixes and improvements • Fixed an issue that caused shadow manager to not recognize the `delta` property when syncing shadow states from AWS IoT Core. • Fixed an issue that sometimes caused sync requests for a shadow to be merged incorrectly.
2.0.1	Version updated for Greengrass nucleus version 2.3.0 release.
2.0.0	Initial version.

Amazon SNS

The Amazon SNS component (`aws.greengrass.SNS`) publishes messages to an Amazon Simple Notification Service (Amazon SNS) topic. You can use this component to send events from Greengrass core devices to web servers, email addresses, and other message subscribers. For more information, see What is Amazon SNS? in the *Amazon Simple Notification Service Developer Guide*.

To publish to an Amazon SNS topic with this component, publish a message to the topic where this component subscribes. By default, this component subscribes to the `sns/message` local publish/subscribe topic. You can specify other topics, including AWS IoT Core MQTT topics, when you deploy this component.

In your custom component, you might want to implement filtering or formatting logic to process messages from other sources before you publish them to this component. This enables you to centralize your message processing logic on a single component.

Note

This component provides similar functionality to the Amazon SNS connector in AWS IoT Greengrass V1. For more information, see Amazon SNS connector in the *AWS IoT Greengrass V1 Developer Guide*.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Input data
- Output data
- Local log file
- Licenses
- Changelog

Versions

This component has the following versions:

- 2.1.x
- 2.0.x

Type

This component is a Lambda component (`aws.greengrass.lambda`). The Greengrass nucleus runs this component's Lambda function using the Lambda launcher component.

For more information, see Component types.

Operating system

This component can be installed on Linux core devices only.

Requirements

This component has the following requirements:

- Your core device must meet the requirements to run Lambda functions. If you want the core device to run containerized Lambda functions, the device must meet the requirements to do so. For more information, see Lambda function requirements.
- Python version 3.7 installed on the core device and added to the PATH environment variable.
- An Amazon SNS topic. For more information, see Creating an Amazon SNS topic in the *Amazon Simple Notification Service Developer Guide*.
- The Greengrass device role must allow the `sns:Publish` action, as shown in the following example IAM policy.

  JSON

  ```
  {
    "Version":"2012-10-17",
    "Statement": [
      {
        "Action": [
          "sns:Publish"
        ],
        "Effect": "Allow",
        "Resource": [
          "arn:aws:sns:us-east-1:123456789012:topic-name"
        ]
      }
    ]
  }
  ```

  You can dynamically override the default topic in the input message payload for this component. If your application uses this feature, the IAM policy must include all target topics as resources. You can grant granular or conditional access to resources (for example, by using a wildcard * naming scheme).
- To receive output data from this component, you must merge the following configuration update for the legacy subscription router component

(`aws.greengrass.LegacySubscriptionRouter`) when you deploy this component. This configuration specifies the topic where this component publishes responses.

Legacy subscription router v2.1.x

```
{
  "subscriptions": {
    "aws-greengrass-sns": {
      "id": "aws-greengrass-sns",
      "source": "component:aws.greengrass.SNS",
      "subject": "sns/message/status",
      "target": "cloud"
    }
  }
}
```

Legacy subscription router v2.0.x

```
{
  "subscriptions": {
    "aws-greengrass-sns": {
      "id": "aws-greengrass-sns",
      "source": "arn:aws:lambda:region:aws:function:aws-greengrass-sns:version",
      "subject": "sns/message/status",
      "target": "cloud"
    }
  }
}
```

- Replace *`region`* with the AWS Region that you use.
- Replace *`version`* with the version of the Lambda function that this component runs. To find the Lambda function version, you must view the recipe for the version of this component that you want to deploy. Open this component's details page in the AWS IoT Greengrass console, and look for the **Lambda function** key-value pair. This key-value pair contains the name and version of the Lambda function.

> **Important**
>
> You must update the Lambda function version on the legacy subscription router every time you deploy this component. This ensures that you use the correct Lambda function version for the component version that you deploy.

For more information, see Create deployments.

- The Amazon SNS component is supported to run in a VPC. To deploy this component in a VPC, the following is required.
  - The Amazon SNS component must have connectivity to `sns.region.amazonaws.com` which has the VPC endpoint of `com.amazonaws.us-east-1.sns`.

Endpoints and ports

This component must be able to perform outbound requests to the following endpoints and ports, in addition to endpoints and ports required for basic operation. For more information, see Allow device traffic through a proxy or firewall.

Endpoint	Port	Required	Description
`sns.region.amazonaws.com`	443	Yes	Publish messages to Amazon SNS.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of

the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

2.1.10

The following table lists the dependencies for version 2.1.10 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.16.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.1.9

The following table lists the dependencies for version 2.1.9 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.15.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.1.8

The following table lists the dependencies for version 2.1.8 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.14.0	Hard

Dependency	Compatible versions	Dependency type
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.1.7

The following table lists the dependencies for version 2.1.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.13.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.1.6

The following table lists the dependencies for version 2.1.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.12.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.1.5

The following table lists the dependencies for version 2.1.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.11.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.1.4

The following table lists the dependencies for version 2.1.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.10.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.1.3

The following table lists the dependencies for version 2.1.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.9.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.1.2

The following table lists the dependencies for version 2.1.2 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.8.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.1.1

The following table lists the dependencies for version 2.1.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.7.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.0.8 - 2.1.0

The following table lists the dependencies for versions 2.0.8 and 2.1.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.6.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft

Dependency	Compatible versions	Dependency type
Token exchange service	^2.0.0	Hard

2.0.7

The following table lists the dependencies for version 2.0.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.5.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.0.6

The following table lists the dependencies for version 2.0.6 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.4.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.0.5

The following table lists the dependencies for version 2.0.5 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.3.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.0.4

The following table lists the dependencies for version 2.0.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.2.0	Hard
Lambda launcher	^2.0.0	Hard
Lambda runtimes	^2.0.0	Soft
Token exchange service	^2.0.0	Hard

2.0.3

The following table lists the dependencies for version 2.0.3 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.3 <2.1.0	Hard
Lambda launcher	>=1.0.0	Hard
Lambda runtimes	>=1.0.0	Soft
Token exchange service	>=1.0.0	Hard

For more information about component dependencies, see the component recipe reference.

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

> **Note**
>
> This component's default configuration includes Lambda function parameters. We recommend that you edit only the following parameters to configure this component on your devices.

`lambdaParams`

An object that contains the parameters for this component's Lambda function. This object contains the following information:

`EnvironmentVariables`

An object that contains the Lambda function's parameters. This object contains the following information:

`DEFAULT_SNS_ARN`

The ARN of the default Amazon SNS topic where this component publishes messages. You can override the destination topic with the `sns_topic_arn` property in the input message payload.

`containerMode`

(Optional) The containerization mode for this component. Choose from the following options:

- `NoContainer` – The component doesn't run in an isolated runtime environment.
- `GreengrassContainer` – The component runs in an isolated runtime environment inside the AWS IoT Greengrass container.

Default: `GreengrassContainer`

`containerParams`

(Optional) An object that contains the container parameters for this component. The component uses these parameters if you specify `GreengrassContainer` for `containerMode`.

This object contains the following information:

`memorySize`

(Optional) The amount of memory (in kilobytes) to allocate to the component.

Defaults to 512 MB (525,312 KB).

`pubsubTopics`

(Optional) An object that contains the topics where the component subscribes to receive messages. You can specify each topic and whether the component subscribes to MQTT topics from AWS IoT Core or local publish/subscribe topics.

This object contains the following information:

`0` – This is an array index as a string.

An object that contains the following information:

`type`

(Optional) The type of publish/subscribe messaging that this component uses to subscribe to messages. Choose from the following options:

- PUB_SUB – Subscribe to local publish/subscribe messages. If you choose this option, the topic can't contain MQTT wildcards. For more information about how to send messages from custom component when you specify this option, see Publish/subscribe local messages.
- `IOT_CORE` – Subscribe to AWS IoT Core MQTT messages. If you choose this option, the topic can contain MQTT wildcards. For more information about how to send messages from custom components when you specify this option, see Publish/subscribe AWS IoT Core MQTT messages.

Default: PUB_SUB

`topic`

(Optional) The topic to which the component subscribes to receive messages. If you specify `IotCore` for `type`, you can use MQTT wildcards (+ and #) in this topic.

Example: Configuration merge update (container mode)

```
{
```

```
  "lambdaExecutionParameters": {
    "EnvironmentVariables": {
      "DEFAULT_SNS_ARN": "arn:aws:sns:us-west-2:123456789012:mytopic"
    }
  },
  "containerMode": "GreengrassContainer"
}
```

Example: Configuration merge update (no container mode)

```
{
  "lambdaExecutionParameters": {
    "EnvironmentVariables": {
      "DEFAULT_SNS_ARN": "arn:aws:sns:us-west-2:123456789012:mytopic"
    }
  },
  "containerMode": "NoContainer"
}
```

Input data

This component accepts messages on the following topic and publishes the message as is to the target Amazon SNS topic. By default, this component subscribes to local publish/subscribe messages. For more information about how to publish messages to this component from your custom components, see Publish/subscribe local messages.

Default topic (local publish/subscribe): `sns/message`

The message accepts the following properties. Input messages must be in JSON format.

`request`

The information about the message to send to the Amazon SNS topic.

Type: `object` that contains the following information:

`message`

The content of the message as a string.

To send a JSON object, serialize it as a string, and specify `json` for the `message_structure` property.

Type: `string`

`subject`

(Optional) The subject of the message.

Type: `string`

The subject can be ASCII text and up to 100 characters. It must begin with a letter, number, or punctuation mark. It can't include line breaks or control characters.

`sns_topic_arn`

(Optional) The ARN of the Amazon SNS topic where this component publishes the message. Specify this property to override the default Amazon SNS topic.

Type: `string`

`message_structure`

(Optional) The structure of the message. Specify `json` to send a JSON message that you serialize as a string in the `content` property.

Type: `string`

Valid values: `json`

`id`

An arbitrary ID for the request. Use this property to map an input request to an output response. When you specify this property, the component sets the `id` property in the response object to this value.

Type: `string`

> **Note**
>
> The message size can be a maximum of 256 KB.

Example input: String message

```
{
  "request": {
```

```
    "subject": "Message subject",
    "message": "Message data",
    "sns_topic_arn": "arn:aws:sns:region:account-id:topic2-name"
  },
  "id": "request123"
}
```

Example input: JSON message

```
{
  "request": {
    "subject": "Message subject",
    "message": "{ \"default\": \"Message data\" }",
    "message_structure": "json"
  },
  "id": "request123"
}
```

Output data

This component publishes responses as output data on the following MQTT topic by default. You must specify this topic as the `subject` in the configuration for the legacy subscription router component. For more information about how to subscribe to messages on this topic in your custom components, see Publish/subscribe AWS IoT Core MQTT messages.

Default topic (AWS IoT Core MQTT): `sns/message/status`

Example output: Success

```
{
  "response": {
    "sns_message_id": "f80a81bc-f44c-56f2-a0f0-d5af6a727c8a",
    "status": "success"
  },
  "id": "request123"
}
```

Example output: Failure

```
{
  "response" : {
    "error": "InvalidInputException",
```

```
    "error_message": "SNS Topic Arn is invalid",
    "status": "fail"
  },
  "id": "request123"
}
```

Local log file

This component uses the following log file.

```
/greengrass/v2/logs/aws.greengrass.SNS.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* with the path to the AWS IoT Greengrass root folder.

  ```
  sudo tail -f /greengrass/v2/logs/aws.greengrass.SNS.log
  ```

Licenses

This component includes the following third-party software/licensing:

- AWS SDK for Python (Boto3)/Apache License 2.0
- botocore/Apache License 2.0
- dateutil/PSF License
- docutils/BSD License, GNU General Public License (GPL), Python Software Foundation License, Public Domain
- jmespath/MIT License
- s3transfer/Apache License 2.0
- urllib3/MIT License

This component is released under the Greengrass Core Software License Agreement.

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.1.10	Version updated for Greengrass nucleus version 2.15.0 release.
2.1.9	Version updated for Greengrass nucleus version 2.14.0 release.
2.1.8	Version updated for Greengrass nucleus version 2.13.0 release.
2.1.7	Version updated for Greengrass nucleus version 2.12.0 release.
2.1.6	Version updated for Greengrass nucleus version 2.11.0 release.
2.1.5	Version updated for Greengrass nucleus version 2.10.0 release.
2.1.4	Version updated for Greengrass nucleus version 2.9.0 release.
2.1.3	Version updated for Greengrass nucleus version 2.8.0 release.
2.1.2	Version updated for Greengrass nucleus version 2.7.0 release.
2.1.1	Version updated for Greengrass nucleus version 2.6.0 release.
2.1.0	New features • Adds support for HTTPS network proxy configurations. For more information, see Connect on port 443 or through a network proxy and Enable the core device to trust an HTTPS proxy.
2.0.8	Version updated for Greengrass nucleus version 2.5.0 release.
2.0.7	Version updated for Greengrass nucleus version 2.4.0 release.
2.0.6	Version updated for Greengrass nucleus version 2.3.0 release.
2.0.5	Version updated for Greengrass nucleus version 2.2.0 release.
2.0.4	Version updated for Greengrass nucleus version 2.1.0 release.
2.0.3	Initial version.

Stream manager

The stream manager component (`aws.greengrass.StreamManager`) enables you to process data streams to transfer to the AWS Cloud from Greengrass core devices.

For more information about how to configure and use stream manager in custom components, see Manage data streams on Greengrass core devices.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Local log file
- Changelog

Versions

This component has the following versions:

- 2.3.x
- 2.2.x
- 2.1.x
- 2.0.x

> **Note**
>
> If you use stream manager to export data to the cloud, you can't upgrade version 2.0.7 of the stream manager component to a version between v2.0.8 and v2.0.11. If you are deploying stream manager for the first time, we strongly recommend that you deploy the latest version of the stream manager component.

Type

This component is a generic component (`aws.greengrass.generic`). The Greengrass nucleus runs the component's lifecycle scripts.

For more information, see Component types.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

Requirements

This component has the following requirements:

- The token exchange role must allow access to the AWS Cloud destinations that you use with stream manager. For more information, see:
  - the section called “AWS IoT Analytics channels (discontinued)”
  - the section called “Amazon Kinesis data streams”
  - the section called “AWS IoT SiteWise asset properties”
  - the section called “Amazon S3 objects”
- The stream manager component is supported to run in a VPC. To deploy this component in a VPC, the following is required.
  - The stream manager component must have connectivity to the AWS service you publish data to.
    - Amazon S3: `com.amazonaws.`*`region`*`.s3`
    - Amazon Kinesis Data Streams: `com.amazonaws.`*`region`*`.kinesis-streams`
    - AWS IoT SiteWise: `com.amazonaws.`*`region`*`.iotsitewise.data`
  - If you publish data to Amazon S3 in the `us-east-1` region, this component will attempt to use the S3 global endpoint by default; however, this endpoint is not available through the Amazon S3 VPC interface endpoint. For more information, see Restrictions and limitations of AWS PrivateLink for Amazon S3. To resolve this, you can choose from the following options.

- Configure the stream manager component to use the regional S3 endpoint in the `us-east-1` region, by setting up `-Daws.s3UseUsEast1RegionalEndpoint=regional` in `JVM_ARGS`.
- Create an Amazon S3 gateway VPC endpoint instead of an Amazon S3 interface VPC endpoint. S3 gateway endpoints support access to the S3 global endpoint. For more information, see Create a gateway endpoint.

Endpoints and ports

This component must be able to perform outbound requests to the following endpoints and ports, in addition to endpoints and ports required for basic operation. For more information, see Allow device traffic through a proxy or firewall.

Endpoint	Port	Required	Description
`kinesis.region.amazonaws.com`	443	No	Required if you publish data to Firehose.
`data.iotsitewise.region.amazonaws.com`	443	No	Required if you publish data to AWS IoT SiteWise.
`*.s3.amazonaws.com`	443	No	Required if you publish data to S3 buckets. You can replace

Endpoint	Port	Required	Description
			* with the name of each bucket where you publish data.

AWS IoT Analytics discontinued

AWS IoT Analytics was discontinued on December 15, 2025. The `iotanalytics.region.amazonaws.com` endpoint is no longer functional. If you have streams configured with `IoTAnalyticsConfig` export destinations, remove them to avoid continuous failed export attempts. For more information, see the section called "AWS IoT Analytics channels (discontinued)".

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

2.1.3 – 2.3.1

The following table lists the dependencies for versions 2.1.3 to 2.3.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <3.0.0	Soft

Dependency	Compatible versions	Dependency type
Token exchange service	>=0.0.0	Hard

2.1.11 – 2.1.12

The following table lists the dependencies for version 2.1.11 to 2.1.10 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.13.0	Soft
Token exchange service	>=0.0.0	Hard

2.1.9 – 2.1.10

The following table lists the dependencies for versions 2.1.9 to 2.1.10 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.12.0	Soft
Token exchange service	>=0.0.0	Hard

2.1.5 – 2.1.8

The following table lists the dependencies for versions 2.1.5 to 2.1.8 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.11.0	Soft
Token exchange service	>=0.0.0	Hard

2.1.2 – 2.1.4

The following table lists the dependencies for versions 2.1.2 to 2.1.4 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.10.0	Soft
Token exchange service	>=0.0.0	Hard

2.1.1

The following table lists the dependencies for version 2.1.1 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.9.0	Soft
Token exchange service	>=0.0.0	Hard

2.1.0

The following table lists the dependencies for version 2.1.0 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.8.0	Soft
Token exchange service	>=0.0.0	Hard

2.0.15

The following table lists the dependencies for version 2.0.15 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.7.0	Soft
Token exchange service	>=0.0.0	Hard

2.0.13 and 2.0.14

The following table lists the dependencies for versions 2.0.13 and 2.0.14 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.6.0	Soft
Token exchange service	>=0.0.0	Hard

2.0.11 and 2.0.12

The following table lists the dependencies for versions 2.0.11 and 2.0.12 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.5.0	Soft
Token exchange service	>=0.0.0	Hard

2.0.10

The following table lists the dependencies for version 2.0.10 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.4.0	Soft
Token exchange service	>=0.0.0	Hard

2.0.9

The following table lists the dependencies for version 2.0.9 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.3.0	Soft

Dependency	Compatible versions	Dependency type
Token exchange service	>=0.0.0	Hard

2.0.8

The following table lists the dependencies for version 2.0.8 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.0 <2.2.0	Soft
Token exchange service	>=0.0.0	Hard

2.0.7

The following table lists the dependencies for version 2.0.7 of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.0.3 <2.1.0	Soft
Token exchange service	>=0.0.0	Hard

For more information about component dependencies, see the component recipe reference.

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

`STREAM_MANAGER_STORE_ROOT_DIR`

(Optional) The absolute path of the local directory used to store streams. This value must start with a forward slash (for example, `/data`).

You must specify an existing folder, and the system user who runs the stream manager component must have permissions to read and write to this folder. For example, you can run

the following commands to create and configure a folder, `/var/greengrass/streams`, which you specify as the stream manager root folder. These commands allow the default system user, `ggc_user`, to read and write to this folder.

```
sudo mkdir /var/greengrass/streams
sudo chown ggc_user /var/greengrass/streams
sudo chmod 700 /var/greengrass/streams
```

Default: *`/greengrass/v2`*`/work/aws.greengrass.StreamManager`

`STREAM_MANAGER_SERVER_PORT`

(Optional) The local port number to use to communicate with stream manager.

You can specify `0` to use a random available port.

Default: `8088`

`STREAM_MANAGER_AUTHENTICATE_CLIENT`

(Optional) You can make it mandatory for clients to authenticate before they can interact with stream manager. The Stream Manager SDK controls interaction between clients and stream manager. This parameter determines which clients can call the Stream Manager SDK to work with streams. For more information, see stream manager client authentication.

If you specify `true`, the Stream Manager SDK allows only Greengrass components as clients.

If you specify `false`, the Stream Manager SDK allows all processes on the core device to be clients.

Default: `true`

`STREAM_MANAGER_EXPORTER_MAX_BANDWIDTH`

(Optional) The average maximum bandwidth (in kilobits per second) that stream manager can use to export data.

Default: No limit

`STREAM_MANAGER_EXPORTER_THREAD_POOL_SIZE`

(Optional) The maximum number of active threads that stream manager can use to export data.

The optimal size depends on your hardware, stream volume, and planned number of export streams. If your export speed is slow, you can adjust this setting to find the optimal size for your hardware and business case. The CPU and memory of your core device hardware are limiting factors. To start, you might try setting this value equal to the number of processor cores on the device.

Be careful not to set a size that's higher than your hardware can support. Each stream consumes hardware resources, so try to limit the number of export streams on constrained devices.

Default: 5 threads

`STREAM_MANAGER_EXPORTER_S3_DESTINATION_MULTIPART_UPLOAD_MIN_PART_SIZE_BYTES`

(Optional) The minimum size (in bytes) of a part in a multipart upload to Amazon S3. Stream manager uses this setting and the size of the input file to determine how to batch data in a multipart PUT request.

> **Note**
>
> Stream manager uses the streams `sizeThresholdForMultipartUploadBytes` property to determine whether to export to Amazon S3 as a single or multipart upload. AWS IoT Greengrass components can set this threshold when they create a stream that exports to Amazon S3.

Default: `5242880` (5 MB). This is also the minimum value.

`STREAM_MANAGER_EXPORTER_KINESIS_INTER_CALL_DELAY_MS`

(Optional) The delay, in milliseconds, that stream manager waits between consecutive requests when it exports streams to Kinesis Data Streams destinations. This delay applies only when stream manager cannot determine the shard layout of the target data stream. This can happen when the Greengrass token exchange role does not include the `kinesis:ListShards` permission. When stream manager can determine the shard layout, it uses shard-aware pacing instead of this delay. To turn off the fallback delay, set this parameter to `0`.

Default: `1000` (1 second)

`LOG_LEVEL`

(Optional) The logging level for the component. Choose from the following log levels, listed here in level order:

- TRACE
- DEBUG
- INFO
- WARN
- ERROR

Default: INFO

JVM_ARGS

(Optional) The custom Java Virtual Machine arguments to pass to stream manager at startup. Separate multiple arguments by spaces.

Use this parameter only when you must override the default settings used by the JVM. For example, you might need to increase the default heap size if you plan to export a large number of streams.

startupTimeoutSeconds

(Optional) The maximum of time in seconds for the component to start. The component's state changes to ERRORED if it exceeds this timeout.

Default: 120

Example: Configuration merge update

The following example configuration specifies to use a non-default port.

```
{
  "STREAM_MANAGER_SERVER_PORT": "18088"
}
```

Local log file

This component uses the following log file.

Linux

```
/greengrass/v2/logs/aws.greengrass.StreamManager.log
```

Windows

```
C:\greengrass\v2\logs\aws.greengrass.StreamManager.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/aws.greengrass.StreamManager.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\aws.greengrass.StreamManager.log -Tail 10 -
  Wait
  ```

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.3.1	Bug fixes and improvements • Improves the throughput of stream manager exports to Kinesis Data Streams destinations, especially for small record sizes. • Adds shard-aware pacing that distributes records across the shards of the target data stream. This pacing keeps requests within the per-shard limits of Kinesis Data Streams to reduce throttling. To use shard-aware pacing, grant the `kinesis:ListShards` permission in the Greengrass token exchange role. Without this permission, stream manager assigns a random partition key to each record. • Adds the `STREAM_MANAGER_EXPORTER_KINESIS_INTER_CALL_DELAY_MS` configuration parameter, which sets a fallback

Version	Changes
	delay between export requests when stream manager cannot determine the shard layout of the target data stream.
2.3.0	Bug fixes and improvements • Updates to the AWS SDK for Java v2 for improved performance and compatibility. • Changes proxy precedence behavior to align with standard AWS SDK proxy configuration.
2.2.1	Bug fixes and improvements • Fixes an issue where stream manager fails to export messages to Kinesis Data Streams destinations. • Additional improvements to the performance of the stream manager exports to Kinesis Data Streams destinations.
2.2.0	New features • Adds a new configuration key for startup timeout. Default value is 120 seconds. • Adds recipe supports for Greengrass nucleus lite.
2.1.13	Bug fixes and improvements Supports FIPS endpoints for AWS IoT SiteWise
2.1.12	Bug fixes and improvements Updates the order that credentials are used so that Greengrass credentia ls are preferred for AWS service requests.
2.1.11	Version updated for Greengrass nucleus version 2.12.0 release.
2.1.10	Bug fixes and improvements Fixes an issue where the HTTPS proxy configuration doesn't trust the Greengrass certificate authority (CA) certificate chain.
2.1.9	Version updated for Greengrass nucleus version 2.11.0 release.

Version	Changes
2.1.8	Bug fixes and improvements Fixes an issue where stream manager infinitely retries SiteWise exports failing with `InvalidRequestException` .
2.1.7	Bug fixes and improvements Fixes an issue where stream manager fails to read the proxy configura tion correctly.
2.1.6	Bug fixes and improvements Fixes an issue that could cause a crash at startup on certain ARMv8 processors, including the Jetson Nano.
2.1.5	Version updated for Greengrass nucleus version 2.10.0 release.
2.1.4	Bug fixes and improvements • Fixes an issue where entries for the same property asset with the same timestamp within a single batch return `ConflictingOperati onException` from the SiteWise API which causes stream manager to continuously retry. • Updates default connection timeout from 3 seconds to 1 minute.
2.1.3	Bug fixes and improvements Fixes a startup issue on Windows OS when running as the SYSTEM user.
2.1.2	Bug fixes and improvements • Fixes an issue on Windows OS that use a non-English language. • Version updated for Greengrass nucleus version 2.9.0 release.
2.1.1	Version updated for Greengrass nucleus version 2.8.0 release.

Version	Changes
2.1.0	New features • Updates this component to automatically send telemetry metrics to Amazon EventBridge. For more information, see Gather system health telemetry data from AWS IoT Greengrass core devices. This feature requires v2.7.0 or later of the Greengrass nucleus component. • Version updated for Greengrass nucleus version 2.7.0 release.
2.0.15	Version updated for Greengrass nucleus version 2.6.0 release.
2.0.14	This version contains bug fixes and improvements.
2.0.13	Version updated for Greengrass nucleus version 2.5.0 release.
2.0.12	Bug fixes and improvements Fixes an issue that prevented upgrading stream manager v2.0.7 to a version between v2.0.8 and v2.0.11. If you use stream manager to export data to the cloud, you can now upgrade to v2.0.12.
2.0.11	Version updated for Greengrass nucleus version 2.4.0 release.
2.0.10	Version updated for Greengrass nucleus version 2.3.0 release.
2.0.9	Version updated for Greengrass nucleus version 2.2.0 release.
2.0.8	Version updated for Greengrass nucleus version 2.1.0 release.
2.0.7	Initial version.

System log forwarder

The System log forwarder (`aws.greengrass.SystemLogForwarder`) uploads active system logs directly to Amazon CloudWatch using the CloudWatch HTTPS API.

Important

This component will only forward systemd-journald logs generated during runtime. For more information on systemd-journald logs, see systemd-journald and journalctl.

Note

This component requires specific permissions to create and manage CloudWatch log groups and streams.

Topics

- Versions
- Type
- Operating system
- Requirements
- Endpoints and ports
- Dependencies
- Configuration
- Changelog

Versions

This component has the following versions:

- 2.1.x
- 2.0.x

Type

This component is a generic component (`aws.greengrass.generic`). The Greengrass nucleus runs the component's lifecycle scripts.

For more information, see Component types.

Operating system

This component must be installed on systemd-based Linux systems.

Requirements

This component has the following requirements:

The component requires access to create log and stream groups in CloudWatch as well as permission to perform the PutLogEvents HTTP call. You must, at minimum, add the following policy permissions to your Greengrass device's role alias:

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
    {
      "Effect": "Allow",
      "Action": ["logs:CreateLogGroup"],
      "Resource": "arn:aws:logs:us-east-1:111122223333:log-group:greengrass/
systemLogs:*"
    },
    {
      "Effect": "Allow",
      "Action": ["logs:CreateLogStream", "logs:PutLogEvents"],
      "Resource": "arn:aws:logs:us-east-1:111122223333:log-group:greengrass/
systemLogs:log-stream:${credentials-iot:ThingName}"
    }
  ]
}
```

Note

For more information, see the System Log Forwarder Github page.

Endpoints and ports

This component must be able to perform outbound requests to the following endpoints and ports, in addition to endpoints and ports required for basic operation. For more information, see Allow device traffic through a proxy or firewall.

Endpoint	Port	Required	Description
`logs.region.amazonaws.com`	443	No	Required if you write logs to CloudWatch Logs.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

2.1.x

The following table lists the dependencies for version 2.1.x of this component.

Dependency	Compatible versions	Dependency type
Token exchange service	>=2.0.0	Hard
Greengrass nucleus lite	>=2.3.0	Soft

2.0.x

The following table lists the dependencies for version 2.0.x of this component.

Dependency	Compatible versions	Dependency type
Token exchange service	>=2.0.0	Hard

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

2.0.x-2.1.x

`maxUploadIntervalSec`

The maximum period at which system log forwarder will attempt to upload logs. Since log forwarder will upload logs when the memory fills, it may still upload more often than the configured maximum cadence.

`maxRetriesCount`

Number of times system log forwarder will attempt to retry a transient HTTP error.

`bufferCapacity`

The size of the ring buffer for in-memory log storage.

`logGroup`

The log path in CloudWatch.

`logStream`

The CloudWatch logStream.

`filters`

A map of filter configurations for the core device.

`services`

A list of service name filters that System Log Forwarder will use to determine which logs gets uploaded. A log will only be uploaded if the service it originated from matches at least one of the filters in this list. The filters in this list may either be a string that the service name must fully match, or a string ending with *, which means the prefix must match.

Default: `[ggl.*]`

> **Important**
>
> A log will only be uploaded if the service it originated from matches at least one of the filters in this list.

> **Note**
>
> Using the value * will include all available services.

Example configuration:

The example below will filter logs by all services included in Greengrass Nucleus Lite.

```
{
    "maxUploadIntervalSec": 300,
    "maxRetriesCount": 3,
    "bufferCapacity": 1048576,
    "logGroup": "greengrass/systemLogs",
    "logStream": "deviceName",
    "filters": {
        "services": ["ggl.*"]
    }
}
```

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.1.0	Bug fixes and improvements • Updates the component recipe to properly support Greengrass nucleus. • Improved logging output when there are no logs to upload.

Version	Changes
	• General bug fixes and improvements.
2.0.1	Bug fixes and improvements Updates the component recipe to properly support aarch64 (arm64) systems.
2.0.0	Initial version.

Systems Manager Agent

The AWS Systems Manager Agent component (`aws.greengrass.SystemsManagerAgent`) installs the Systems Manager Agent, so you can manage core devices with Systems Manager. Systems Manager is an AWS service that you can use to view and control your infrastructure on AWS, including Amazon EC2 instances, on-premises servers and virtual machines (VMs), and edge devices. Systems Manager enables you to view operational data, automate operation tasks, and maintain security and compliance. For more information, see What is AWS Systems Manager? and About Systems Manager Agent in the *AWS Systems Manager User Guide*.

Systems Manager tools and features are called *capabilities*. Greengrass core devices support all Systems Manager capabilities. For more information about these capabilities and how to use Systems Manager to manage core devices, see Systems Manager capabilities in the *AWS Systems Manager User Guide*.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Local log file
- See also
- Changelog

Versions

This component has the following versions:

- 1.3.x
- 1.2.x
- 1.1.x
- 1.0.x

Type

This component is a generic component (`aws.greengrass.generic`). The Greengrass nucleus runs the component's lifecycle scripts.

For more information, see Component types.

Operating system

This component can be installed on Linux core devices only.

Requirements

This component has the following requirements:

- A Greengrass core device that runs on a 64-bit Linux platform: Armv8 (AArch64) or x86_64.
- You must have an AWS Identity and Access Management (IAM) service role that Systems Manager can assume. This role must include the AmazonSSMManagedInstanceCore managed policy or a custom policy that defines equivalent permissions. For more information, see Create an IAM service role for edge devices in the *AWS Systems Manager User Guide*.

  When you deploy this component, you must specify this role's name for the `SSMRegistrationRole` configuration parameter.
- The Greengrass device role must allow the `ssm:AddTagsToResource` and `ssm:RegisterManagedInstance` actions. The device role must also allow the `iam:PassRole` action for the IAM service role that fulfills the previous requirement. The following example IAM policy grants these permissions.

```
{
```

```
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "iam:PassRole"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:iam::account-id:role/SSMServiceRole"
      ]
    },
    {
      "Action": [
        "ssm:AddTagsToResource",
        "ssm:RegisterManagedInstance"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

- On Ubuntu devices using snap packaging, this component requires the `core22` base snap. If your device can reach `api.snapcraft.io`, `snap` downloads `core22` automatically. On restricted networks where the device cannot reach the snap store, manually install `core22` before you deploy this component:

  1. Download the snap and its assertion from an internet-connected machine that has the same architecture as the target device:

     ```
     snap download core22
     ```

  2. Transfer the `.snap` and `.assert` files to the target device.

  3. Acknowledge the assertion on the target device:

     ```
     sudo snap ack core22.assert
     ```

  4. Install the snap on the target device:

     ```
     sudo snap install core22.snap
     ```

Endpoints and ports

This component must be able to perform outbound requests to the following endpoints and ports, in addition to endpoints and ports required for basic operation. For more information, see Allow device traffic through a proxy or firewall.

Endpoint	Port	Required	Description
`ec2messages.` *region* `.amazonaws.com`	443	Yes	Communicate with the Systems Manager service in the AWS Cloud.
`ssm.`*region*`.amazonaws.com`	443	Yes	Register the core device as a Systems Manager managed node.
`ssmmessages.` *region* `.amazonaws.com`	443	Yes	Communicate with Session Manager, a capability of Systems Manager, in the AWS Cloud.

For more information, see Reference: ec2messages, ssmmessages, and other API calls in the *AWS Systems Manager User Guide*.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

The following table lists the dependencies for versions 1.0.0 to 1.3.0 of this component.

Dependency	Compatible versions	Dependency type
Token exchange service	>=2.0.0 <3.0.0	Hard

For more information about component dependencies, see the component recipe reference.

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

`SSMRegistrationRole`

The IAM service role that Systems Manager can assume and that includes the AmazonSSMManagedInstanceCore managed policy or a custom policy that defines equivalent permissions. For more information, see Create an IAM service role for edge devices in the *AWS Systems Manager User Guide*.

`SSMOverrideExistingRegistration`

(Optional) If the core device already runs the Systems Manager Agent registered with a hybrid activation, you can override the device's existing Systems Manager Agent registration. Set this option to `true` to register the core device as a managed node using the Systems Manager Agent that this component provides.

> **Note**
>
> This option applies only to devices that are registered with a hybrid activation. If the core device runs on an Amazon EC2 instance with the Systems Manager Agent installed and an instance profile role configured, the Amazon EC2 instance's existing managed node ID starts with `i-`. When you install the Systems Manager Agent component, the Systems Manager agent registers a new managed node whose ID starts with `mi-` instead of `i-`. Then, you can use the managed node whose ID starts with `mi-` to manage the core device with Systems Manager.

Default: `false`

`SSMResourceTags`

(Optional) The tags to add to the Systems Manager managed node that this component creates for the core device. You can use these tags to manage groups of core devices with Systems Manager. For example, you can run a command on all devices that have a tag that you specify.

Specify a list where each tag is an object with a `Key` and a `Value`. For example, the following value for `SSMResourceTags` instructs this component to set the **`Owner`** tag to **`richard-roe`** on the core device's managed node.

```
[
  {
    "Key": "Owner",
    "Value": "richard-roe"
  }
]
```

This component ignores these tags if the managed node already exists and `SSMOverrideExistingRegistration` is `false`.

Example: Configuration merge update

The following example configuration specifies to use a service role named `SSMServiceRole` to allow the core device to register and communicate with Systems Manager.

```
{
```

```
  "SSMRegistrationRole": "SSMServiceRole",
  "SSMOverrideExistingRegistration": false,
  "SSMResourceTags": [
    {
      "Key": "Owner",
      "Value": "richard-roe"
    },
    {
      "Key": "Team",
      "Value": "solar"
    }
  ]
}
```

Local log file

The Systems Manager Agent software writes logs to a folder outside the Greengrass root folder. For more information, see Viewing Systems Manager Agent logs in the *AWS Systems Manager User Guide*.

The Systems Manager Agent component uses shell scripts to install, start, and stop the Systems Manager Agent. You can find the output from these scripts in the following log file.

```
/greengrass/v2/logs/aws.greengrass.SystemsManagerAgent.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* with the path to the AWS IoT Greengrass root folder.

  ```
  sudo tail -f /greengrass/v2/logs/aws.greengrass.SystemsManagerAgent.log
  ```

See also

- Manage Greengrass core devices with AWS Systems Manager
- What is AWS Systems Manager? in the *AWS Systems Manager User Guide*
- About Systems Manager Agent in the *AWS Systems Manager User Guide*

Changelog

The following table describes the changes in each version of the component.

Version	Changes
1.3.0	Bug fixes and improvments Updates this component to get the Agent version 3.3.1142.0.
1.2.4	Bug fixes and improvments Updates this component to get the Agent version 3.2.2303.0.
1.2.3	Bug fixes and improvements • Adds retries for the Agent component's installation with snap on Greengrass. • Updates the Agent component's configuration to use only the Onprem Identity in Greengrass. • Updates this component to update the Agent only when the installed Agent version doesn't match the Greengrass SSM Agent component's version.
1.1.0	This version contains bug fixes and improvements.
1.0.0	Initial version.

Token exchange service

The token exchange service component (`aws.greengrass.TokenExchangeService`) provides AWS credentials that you can use to interact with AWS services in your custom components.

The token exchange service runs as a local server. This local server connects to the AWS IoT credentials provider using the AWS IoT role alias that you configure in the Greengrass core nucleus component. The component provides two environment variables, `AWS_CONTAINER_CREDENTIALS_FULL_URI` and `AWS_CONTAINER_AUTHORIZATION_TOKEN`. `AWS_CONTAINER_CREDENTIALS_FULL_URI` defines the URI to this local server. When a component creates an AWS SDK client, the client recognizes this URI environment variable and

uses the token in the `AWS_CONTAINER_AUTHORIZATION_TOKEN` to connect to the token exchange service and retrieve AWS credentials. This allows Greengrass core devices to call AWS service operations. For more information about how to use this component in custom components, see Interact with AWS services.

Important

Support to acquire AWS credentials in this way was added to the AWS SDKs on July 13th, 2016. Your component must use an AWS SDK version that was created on or after that date. For more information, see Using a supported AWS SDK in the *Amazon Elastic Container Service Developer Guide*.

Note

Components using the AWS Default Credential Chain may discover and use other sources of credentials, if they exist. Within the Amazon Amazon Elastic Container Service (Amazon ECS) container credentials `AWS_CONTAINER_CREDENTIALS_RELATIVE_URI` will take precedence over `AWS_CONTAINER_CREDENTIALS_FULL_URI`, meaning that token exchange service credentials may not work within Amazon Elastic Container Service (Amazon ECS). For more, see standardized credential providers

Topics

- Versions
- Type
- Operating system
- Dependencies
- Configuration
- Local log file
- Changelog

Versions

This component has the following versions:

- 2.0.x

Type

This component is a generic component (`aws.greengrass.generic`). The Greengrass nucleus runs the component's lifecycle scripts.

For more information, see Component types.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

Dependencies

This component doesn't have any dependencies.

Configuration

This component provides the following configuration parameters that you can customize when you deploy the component.

`port`

The port to use for token exchange service connections. The token exchange service will restart after port configuration changes.

`credentialRetryInSec`

Specifies retry intervals in seconds when Token Exchange Service encounters credential request errors.

`clientError`

The retry interval in seconds for client errors (4xx HTTP status codes).

Default: `120`

Valid values: `10` to `42900`

`serverError`

The retry interval in seconds for server errors (5xx HTTP status codes).

Default: `60`

Valid values: `10` to `42900`

`unknownError`

The retry interval in seconds for unknown errors (connection errors and HTTP status codes outside the 4xx and 5xx ranges).

Default: `300`

Valid values: `10` to `42900`

Example: Configuration merge update

```
{
   "port": 2020,
   "credentialRetryInSec": {
    "clientError": 30,
    "serverError": 45,
    "unknownError": 60
   }
}
```

Local log file

This component uses the same log file as the Greengrass nucleus component.

Linux

```
/greengrass/v2/logs/greengrass.log
```

Windows

```
C:\greengrass\v2\logs\greengrass.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
  ```

Changelog

The following table describes the changes in each version of the component.

Version	Changes
2.0.3	Initial version.

IoT SiteWise OPC UA collector

The IoT SiteWise OPC UA collector component (`aws.iot.SiteWiseEdgeCollectorOpcua`) enables AWS IoT SiteWise gateways to collect data from local OPC UA servers.

With this component, AWS IoT SiteWise gateways can connect to multiple OPC UA servers. For more information about AWS IoT SiteWise gateways, see Using AWS IoT SiteWise at the edge in the *AWS IoT SiteWise User Guide*.

Topics

- Versions
- Type
- Operating system
- Requirements

- Dependencies
- Configuration
- Input data
- Output data
- Local log file
- Licenses
- Changelog
- See also

Versions

This component has the following versions:

- 3.1.x
- 3.0.x
- 2.6.x
- 2.5.x
- 2.4.x
- 2.3.x
- 2.2.x
- 2.1.x
- 2.0.x

Type

This component is a generic component (`aws.greengrass.generic`). The Greengrass nucleus runs the component's lifecycle scripts.

For more information, see Component types.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux

- Windows

Requirements

This component has the following requirements:

- The Greengrass core device must run on one of the following platforms:
  - os: Ubuntu 20.04 or later

    architecture: x86_64 (AMD64) or ARMv8 (Aarch64)
  - os: Red Hat Enterprise Linux (RHEL) 8

    architecture: x86_64 (AMD64) or ARMv8 (Aarch64)
  - os: Amazon Linux 2

    architecture: x86_64 (AMD64) or ARMv8 (Aarch64)
  - os: Debian 11

    architecture: x86_64 (AMD64) or ARMv8 (Aarch64)
  - os: Windows Server 2019 or later

    architecture: x86_64 (AMD64)
- The Greengrass core device must allow outbound network connectivity to OPC UA servers.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

The following table lists the dependencies for all versions of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.3.0 <3.0.0	Hard
Stream manager	>2.0.10<3.0.0	Hard
Secret manager	>=2.0.8 <3.0.0	Hard

For more information about component dependencies, see the component recipe reference.

Configuration

This component doesn't have any configuration parameters.

You can use the AWS IoT SiteWise console or API to configure the IoT SiteWise OPC UA collector component. For more information, see Step 4: Add data sources - optional in the *AWS IoT SiteWise User Guide*.

Input data

This component only accepts data in the following formats, all others will be ignored and discarded. The table below maps the OPC UA data types to their SiteWise equivalent.

SiteWise data type	OPC UA data type	Description

`STRING`	`String`
	`Guid`
	`XmlElement`
		A string of maximum length 1024 bytes.

`INTEGER`	`SByte`
	`Byte`
	`Int16`
	`UInt16`
	`Int32`
		A signed 32-bit integer with a range from `-2,147,483,648` to `2,147,483,647`.

SiteWise data type	OPC UA data type	Description
	`UInt32*` `Int64*`	

`DOUBLE`	`UInt32*`
	`Int64*`
	`Float`
	`Double`
		A floating point number with range from `–10^100 to 10^100` and `IEEE 754` double precision.

`BOOLEAN`	`Boolean`	`true` or `false`.

* For OPC UA data types `UInt32` and `Int64`, its SiteWise data type will be `INTEGER` if SiteWise is able to represent its value, otherwise it will be `DOUBLE`.

Output data

This component writes `BatchPutAssetPropertyValue` messages to AWS IoT Greengrass stream manager. For more information, see BatchPutAssetPropertyValue in the *AWS IoT SiteWise API Reference*.

Local log file

This component uses the following log file.

Linux

```
/greengrass/v2/logs/aws.iot.SiteWiseEdgeCollectorOpcua.log
```

Windows

```
C:\greengrass\v2\logs\aws.iot.SiteWiseEdgeCollectorOpcua.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/aws.iot.SiteWiseEdgeCollectorOpcua.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\aws.iot.SiteWiseEdgeCollectorOpcua.log -Tail
   10 -Wait
  ```

Licenses

This component is released under the Greengrass Core Software License Agreement.

Changelog

The following table describes the changes in each version of the component.

Version	Changes
3.1.0	New features • Adds configurability of the node discovery process for each data source. • Adds configurability of the default data change trigger for each data source. Bug fixes and improvements • Fixes security vulnerabilities.
3.0.3	Bug fixes and improvements • Fixes an issue that caused file truncation and sync failure when syncing large configurations.

Version	Changes
	• Fixes an issue that caused startup process delays due to AWS IoT Core subscription latency. • Fixes an issue that triggered full discovery upon momentary disconnections from OPC UA servers.
3.0.2	Bug fixes and improvements • Fixes the `OpcUaCollector.IncomingValuesCount` metric to accurately include snapshot points in calculations.
3.0.1	Bug fixes and improvements • Fixes security vulnerabilities. • Fixes an issue with OPC UA timestamp handling. Adds a preferred timestamp selection and improves fallback logic when the preferred timestamps are unavailable. • Fixes an issue where Classic streams, V2 gateway OPC UA collector configurations fail to default to Stream Manager when no destination is specified.
3.0.0	New features Added support for MQTT-enabled, V3 gateways in addition to Classic streams, V2 gateways (formerly known as self-hosted SiteWise Edge gateways). For more information, see MQTT-enabled, V3 gateways for AWS IoT SiteWise Edge. • Allows OPC UA data to be published to the AWS IoT Greengrass MQTT 5 broker (EMQX). Bug fixes and improvements • Fixes security vulnerabilities.
2.6.0	New features • Added support for ingestion of Null and NaN values if ingestion is enabled in AWS IoT SiteWise. To view or modify the Null and NaN configuration in AWS IoT SiteWise, see the DescribeStorageConfiguration and PutStorageConfiguration APIs.

Version	Changes
2.5.1	Bug fixes and improvements • Fixes a bug where future snapshotting tasks are cancelled if an error is encountered while a snapshot task is running. • Fixes a bug where data source configuration updates do not persist until after restarting the OPC UA Collector, if the connection to the data source's OPC UA server is lost.
2.5.0	New features • Adds a data source option to convert simple arrays and DateTime values to strings. • Adds a property group option to select either a source or server timestamp when collecting data from an OPC UA server. Bug fixes and improvements • Resets the default polling configuration to use the source timestamp.
2.4.2	Bug fixes and improvements • Fixes issues during OPC UA server discovery in which a node may be discovered multiple times. • Fixes the snapshot feature to ensure the timestamp is new for each snapshot data point.
2.4.1	Bug fixes and improvements • Fixes issues related to proxy support. • Fixes issue where thread cleanup failed and caused a data blockage.
2.4.0	New features • Adds an events log to make it easier to identify and remediate problems. Bug fixes and improvements • Fixes an issue with the OPC UA client that caused certificate errors when connecting to an OPC UA server that uses version 1.05 of the OPC UA specification.

Version	Changes
2.3.0	New features • Adds support for the Greengrass nucleus HTTP proxy configuration on Linux. Bug fixes and improvements • Fixes a security issue (CVE-2019-19135).

2.2.0	New features
	• Adds support for installing Data Collection Pack on Linux ARMv8 architecture.
	• Minimum requirements for Linux ARMv8:
	  • Memory: 4 GB
	  • CPU: ARM Cortex-A72 or equivalent specification
	
	Bug fixes and improvements
	• Improves logging of metrics in node discovery process.
	• Improves handling of unsupported data types.
	• Improves logging of data stream errors.

2.1.3	New features • Adds support for Windows Server 2019 or higher. Bug fixes and improvements • Improves error messages when you deploy this component on unsupported devices.

Version	Changes

2.1.1	New features
	• Adds support for configuring the following subscription properties:
	  • DataChangeTrigger - You can define the condition that initiates a data change alert.
	  • QueueSize - The depth of the queue on an OPC-UA server for a particular metric where notifications for Monitored Items are queued.
	  • PublishingIntervalMilliseconds - The interval (in milliseconds) of a publishing cycle specified when a subscription is created.
	  • SnapshotFrequencyMilliseconds - You can configure the snapshot frequency timeout setting to ensure that AWS IoT SiteWise Edge ingests a steady stream of data.
	• This version supports ingestion of BAD quality data and filters data based on the following data qualities:
	  • UNCERTAIN quality data
	  • BAD quality data
	Bug fixes and improvements
	• Improvements to customer metrics.
	• Fixes the security encoding that sometimes caused issues when connecting to servers with encryption enabled.
	• Fixes an issue where the property group failed to update.

2.0.3	Bug fixes and improvements.
2.0.2	Bug fixes and improvements to asset priority syncing with edge.
2.0.1	Initial version.

See also

- What is AWS IoT SiteWise? in the *AWS IoT SiteWise User Guide*.

IoT SiteWise OPC UA data source simulator

The IoT SiteWise OPC UA data source simulator component (`aws.iot.SiteWiseEdgeOpcuaDataSourceSimulator`) starts a local OPC UA server that generates sample data. Use this OPC UA server to simulate a data source read by the IoT SiteWise OPC UA collector component on an AWS IoT SiteWise gateway. Then, you can explore AWS IoT SiteWise features using this sample data. For more information about AWS IoT SiteWise gateways, see Using AWS IoT SiteWise at the edge in the *AWS IoT SiteWise User Guide*.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Local log file
- Changelog
- See also

Versions

This component has the following versions:

- 1.0.x

Type

This component is a generic component (`aws.greengrass.generic`). The Greengrass nucleus runs the component's lifecycle scripts.

For more information, see Component types.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

Requirements

This component has the following requirements:

- The Greengrass core device must be able to use port 4840 on the local host. This component's local OPC UA server runs at this port.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

The following table lists the dependencies for all versions of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.3.0 <3.0.0	Hard

For more information about component dependencies, see the component recipe reference.

Configuration

This component doesn't have any configuration parameters.

Local log file

This component uses the following log file.

Linux

```
/greengrass/v2/logs/aws.iot.SiteWiseEdgeOpcuaDataSourceSimulator.log
```

Windows

```
C:\greengrass\v2\logs\aws.iot.SiteWiseEdgeOpcuaDataSourceSimulator.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/
  aws.iot.SiteWiseEdgeOpcuaDataSourceSimulator.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs
  \aws.iot.SiteWiseEdgeOpcuaDataSourceSimulator.log -Tail 10 -Wait
  ```

Changelog

The following table describes the changes in each version of the component.

Version	Changes
1.0.0	Initial version. Adds support for Windows Server 2016 or higher.

See also

- What is AWS IoT SiteWise? in the *AWS IoT SiteWise User Guide*.

IoT SiteWise publisher

The IoT SiteWise publisher component (`aws.iot.SiteWiseEdgePublisher`) enables AWS IoT SiteWise gateways to export data from the edge to the AWS Cloud.

For more information about AWS IoT SiteWise gateways, see Using AWS IoT SiteWise at the edge in the *AWS IoT SiteWise User Guide*.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Input data
- Local log file
- Troubleshooting and debugging
- Licenses
- Changelog
- See also

Versions

This component has the following versions:

- 4.1.x
- 4.0.x
- 3.2.x
- 3.1.x
- 3.0.x
- 2.4.x
- 2.3.x

- 2.2.x
- 2.1.x
- 2.0.x

Type

This component is a generic component (`aws.greengrass.generic`). The Greengrass nucleus runs the component's lifecycle scripts.

For more information, see Component types.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

Requirements

This component has the following requirements:

- The Greengrass core device must run on one of the following platforms:
  - os: Ubuntu 18.04 or later

    architecture: x86_64 (AMD64) or ARMv8 (Aarch64)
  - os: Red Hat Enterprise Linux (RHEL) 8

    architecture: x86_64 (AMD64) or ARMv8 (Aarch64)
  - os: Amazon Linux 2

    architecture: x86_64 (AMD64) or ARMv8 (Aarch64)
  - os: Debian 11

    architecture: x86_64 (AMD64) or ARMv8 (Aarch64)
  - os: Windows Server 2019 or later

    architecture: x86_64 (AMD64)

- The Greengrass core device must connect to the Internet.
- The Greengrass core device must be authorized to perform the `iotsitewise:BatchPutAssetPropertyValue` action. For more information, see Authorize core devices to interact with AWS services.

  Example permissions policy

  JSON

  ```
  {
      "Version":"2012-10-17",
      "Statement": [
          {
              "Effect": "Allow",
              "Action": "iotsitewise:BatchPutAssetPropertyValue",
              "Resource": "*"
          }
      ]
  }
  ```

Endpoints and ports

This component must be able to perform outbound requests to the following endpoints and ports, in addition to endpoints and ports required for basic operation. For more information, see Allow device traffic through a proxy or firewall.

Endpoint	Port	Required	Description
`data.iotsitewise.` *region* `.amazonaws.com`	443	Yes	Publish data to AWS IoT SiteWise.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of

its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

The following table lists the dependencies for versions 2.0.x to 2.2.x of this component.

Dependency	Compatible versions	Dependency type
Greengrass nucleus	>=2.3.0<3.0.0	Hard
Stream manager	>=2.0.10<3.0.0	Hard

For more information about component dependencies, see the component recipe reference.

Configuration

This component doesn't have any configuration parameters.

You can use the AWS IoT SiteWise console or API to configure the IoT SiteWise publisher component. For more information, see Step 3: Configure publisher - optional in the *AWS IoT SiteWise User Guide*.

Input data

This component reads `PutAssetPropertyValueEntry` messages from AWS IoT Greengrass stream manager. For more information, see PutAssetPropertyValueEntry in the *AWS IoT SiteWise API Reference*.

Local log file

This component uses the following log file.

Linux

```
/greengrass/v2/logs/aws.iot.SiteWiseEdgePublisher.log
```

Windows

```
C:\greengrass\v2\logs\aws.iot.SiteWiseEdgePublisher.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/aws.iot.SiteWiseEdgePublisher.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\aws.iot.SiteWiseEdgePublisher.log -Tail 10 -
  Wait
  ```

Troubleshooting and debugging

This component includes a new events log to help customers identify and fix problems. The log file is separate from the local log file, and is found in the following location. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

Linux

```
/greengrass/v2/work/aws.iot.SiteWiseEdgePublisher/logs/
IotSiteWisePublisherEvents.log
```

Windows

```
C:\greengrass\v2\work\aws.iot.SiteWiseEdgePublisher\logs
\IotSiteWisePublisherEvents.log
```

This log includes detailed information and troubleshooting instructions. Troubleshooting information is provided alongside the diagnostics, with a description of how to remedy the issue, and sometimes with links to further information. Diagnostic information includes the following:

- Severity level
- Timestamp
- Additional event-specific information

Example log

```
accountBeingThrottled:
  Summary: Data upload speed slowed due to quota limits
  Level: WARN
  Timestamp: '2023-06-09T21:30:24.654Z'
  Description: The IoT SiteWise Publisher is limited to the "Rate of data points
 ingested"
    quota for a customers account. See the associated documentation and associated
    metric for the number of requests that were limited for more information. Note
    that this may be temporary and not require any change, although if the issue
 continues
    you may need to request an increase for the mentioned quota.
  FurtherInformation:
  - https://docs.aws.amazon.com/iot-sitewise/latest/userguide/quotas.html
  - https://docs.aws.amazon.com/iot-sitewise/latest/userguide/troubleshooting-
gateway.html#gateway-issue-data-streams
  AssociatedMetrics:
  - Name: TotalErrorCount
    Description: The total number of errors of this type that occurred.
    Value: 327724.0
  AssociatedData:
  - Name: AggregatePropertyAliases
    Description: The aggregated property aliases of the throttled data.
    FileLocation: /greengrass/v2/work/aws.iot.SiteWiseEdgePublisher/./logs/data/
AggregatePropertyAliases_1686346224654.log
```

Licenses

This component is released under the Greengrass Core Software License Agreement.

Changelog

The following table describes the changes in each version of the component.

Version	Changes
4.1.4	Bug fixes and improvements • IoT SiteWise publisher now reports system health metrics, such as memory and CPU usage of the IoT SiteWise publisher, to CloudWatch. • Updated error reporting to AWS IoT Greengrass so that IoT SiteWise publisher is restarted if the startup workflow has partial failure.
4.1.3	Bug fixes and improvements • Fixes a memory leak in the data checkpointing process. • Fixes security vulnerabilities.
4.1.2	Bug fixes and improvements • Fixes an issue involved with syncing large configurations. • Delays in subscription to AWS IoT Core no longer affect the start up process. • Delays in initializing checkpoints no longer affect the start up process.
4.1.1	Bug fixes and improvements • Fixes an issue that causes high CPU and memory usage on Classic streams, V2 gateways when sending data through an AWS IoT SiteWise buffered using Amazon S3 destination after restarting the publisher. • Adds support for overriding default component logging behavior by creating an optional file located at: `/greengrass/v2/work/aws.iot.SiteWiseEdgePublisher/log4j2.xml` .
4.1.0	New features • Adds a command-line tool to manage built-in EMQX MQTT broker users. This interface enables user account management and configure s access control policies for MQTT topic publishing and subscription permissions.

Version	Changes
	Bug fixes and improvements • Fixes the calculation that determines how many unique aliases can fit into a single Buffered ingestion Parquet file.
4.0.3	Bug fixes and improvements • Updates the maximum of unique aliases per AWS IoT SiteWise buffered ingestion file to match the cloud limit of 10,000. • Fixes an AWS IoT Greengrass logging issue that incorrectly reports invalid service states. • Adds validation to prevent duplicate stream names across Amazon S3 destination configurations in Classic streams, V2 gateways. • Fixes a compatibility issue with AWS IoT SiteWise buffered ingestion in non-standard AWS partitions. • Fixes an issue that causes duplicate data ingestion.
4.0.2	Bug fixes and improvements • Fixes an issue that prevents data upload after the batch wait time expires. • Adds validation to verify that each destination in MQTT-enabled, V3 gateway configurations are unique.
4.0.1	Bug fixes and improvements • Fixes an issue that prevents data upload to AWS IoT SiteWise when you disable the maximum batch wait time configuration. • Fixes an issue where the `IoTSiteWisePublisher.NumberOfSubscriptionsToMqttBroker` metric inaccurately reports the count of unique topics subscribed to the MQTT broker.

Version	Changes
4.0.0	New features Adds support for MQTT-enabled, V3 gateways in addition to Classic streams, V2 gateways (formerly known as self-hosted SiteWise Edge gateways). • Adds the ability to subscribe to and receive data from the AWS IoT Greengrass MQTT 5 broker (EMQX). For more information, see MQTT-enabled, V3 gateways for AWS IoT SiteWise Edge. • Adds configuration options for data destinations using path filters. For more information, see Understand path filters for AWS IoT SiteWise Edge destinations. Bug fixes and improvements • Fixes an issue where the `PublishSuccessCount` metric reports negative values. • Fixes an issue where the publisher doesn't start up within 100 seconds and enters the BROKEN state. • Fixes security vulnerabilities.
3.2.0	New features • Added support for ingestion of Null and NaN values if ingestion is enabled in AWS IoT SiteWise. To view or modify the Null and NaN configuration in AWS IoT SiteWise, see the DescribeStorageConfiguration and PutStorageConfiguration APIs. Bug fixes and improvements • Fixes issues causing corrupted checkpoint database files. • Fixes issues generating duplicated metrics.

Version	Changes
3.1.4	**Warning** Version 3.1.4 was discontinued on February 20, 2025. The improvements in this version are available in later versions of this component. Bug fixes and improvements • Fixes issues that could cause longer-than-expected startup times after being offline.

3.1.3	Bug fixes and improvements
	• Fixes an issue where the events log file located at `/greengrass/v2/work/aws.iot.SiteWiseEdgePublisher/logs/IoTSiteWisePublisherEvents.log` was created but no events were being logged.
	• Adds the following CloudWatch metrics for monitoring the connection with the MQTT broker:
	  • `IoTSiteWisePublisher.IsConnectedToMqttBroker`
	  • `IoTSiteWisePublisher.NumberOfSubscriptionsToMqttBroker`
	  • `IoTSiteWisePublisher.NumberOfUniqueMqttTopicsReceived`
	  • `IoTSiteWisePublisher.MqttMessageReceivedSuccessCount`
	  • `IoTSiteWisePublisher.MqttReceivedSuccessBytes`
	  For more information about these metrics, see AWS IoT Greengrass Version 2 gateway metrics.
	• Fixes an issue where the `BatchCreateJob` API is still called even if uploading a parquet file to S3 fails.

Version	Changes
3.1.2	Bug fixes and improvements • Fixes the issue of high CPU usage introduced in version 3.1.1.
3.1.1	**Warning** Version 3.1.1 was discontinued on March 12, 2024. The improvements in this version are available in later versions of this component. Bug fixes and improvements • Adds additional logging that identifies the affected data aliases when an error occurs. • Adds local enforcement of AWS IoT SiteWise API limits on the age of the data ingested. • Fixes the issue where Publisher mixes up the checkpoints of the StreamManager streams when there are multiple Amazon S3 destinations. • Fixes performance bottleneck with how the publisher reads from the StreamManager streams.
3.1.0	New features • Adds support to publish data as parquet files to Amazon S3. • Adds support for AWS IoT SiteWise buffered ingestion.
3.0.0	Bug fixes and improvements • Fixes issues related to proxy support. New features • Enables support of data ingestion from an MQTT broker.

Version	Changes
2.4.1	Bug fixes and improvements • Enable component to work with Java Corretto 11 versions 11.0.20.8.1 and later. Component versions 2.4.0 and 2.3.3 show the `"Could not find or load main class"` error message when used with Java Corretto version 11.0.20.8.1.
2.4.0	New features • Adds a new events log to make it easier to identify and remediate problems. Bug fixes and improvements • Improves Publisher checkpoint recovery.
2.3.3	Bug fixes and improvements • Improves ability to support high throughput.
2.3.2	Bug fixes and improvements • Fixes HTTP proxy support when downloading Publisher configuration.

2.3.1	New features
	• Adds support for installing Data Collection Pack on Linux ARMv8 architecture.
	• Minimum requirements for Linux ARMv8:
	  • Memory: 4 GB
	  • CPU: ARM Cortex-A72 or equivalent specification

2.2.3	Bug fixes and improvements • Removes retry for generic exception which was not in the retriable exception list.
2.2.2	Bug fixes and improvements • Reintroduces data upload support to AWS IoT SiteWise through an HTTP proxy server.

Version	Changes
2.2.1	**Note** This version doesn't support HTTP proxy configuration. Version 2.2.2 and higher reintroduces support for this feature. New features • Adds support to this component to toggle compression when uploading data to AWS IoT SiteWise.

2.2.0	**Note**
	This version doesn't support HTTP proxy configuration. Version 2.2.2 and higher reintroduces support for this feature.
	
	New features
	• Updates this component to compress data before sending it to the AWS IoT SiteWise service.
	  • In most cases, this change reduces bandwidth usage by 75 percent compared to previous versions of this component.
	  • In most cases, this change increases CPU usage by up to 5 percent. On gateways that process large amounts of data, this change can increase CPU usage by up to 15 percent.
	  • This change doesn't affect AWS IoT SiteWise service charges or service quota usage.
	  • Adds support for Windows Server 2019 or higher.
	Bug fixes and improvements
	• Fixes an issue that prevents this component from starting when the checkpoint file is corrupted.

2.1.4	Bug fixes and improvements • Fixes compatibility with Java version 8.

Version	Changes
2.1.3	**Warning** This version is no longer available, except in the US East (Ohio), Canada (Central), and AWS GovCloud (US-East) Regions. This component version requires Java version 11 or greater to run. The improvements in this version are available in later versions of this component. Bug fixes and improvements • Improves error messages when you deploy this component on unsupported devices. • Updates to log errors when data uploads fail.
2.1.2	Bug fixes and improvements • Updates to invoke the expired data export feature as soon as data expires.
2.1.1	Bug fixes and improvements.
2.1.0	New features • Adds support for publishing the newest data to the cloud first. • Adds support for not publishing expired data to the cloud. • Adds support for storing expired data locally. Bug fixes and improvements • Reduces disk I/O and corresponding latency.
2.0.2	Bug fixes and improvements.
2.0.1	Initial version.

See also

- What is AWS IoT SiteWise? in the *AWS IoT SiteWise User Guide*.

IoT SiteWise processor

The IoT SiteWise processor component (`aws.iot.SiteWiseEdgeProcessor`) enables AWS IoT SiteWise Classic streams, V2 gateways to process data at the edge.

With this component, AWS IoT SiteWise gateways can use asset models and assets to process data on gateway devices. For more information about AWS IoT SiteWise gateways, see Using AWS IoT SiteWise at the edge in the *AWS IoT SiteWise User Guide*.

> **Note**
>
> The data processing pack (DPP) feature will no longer be open to new customers starting November 7, 2025. If you would like to use DPP, sign up prior to that date. Existing customers can continue to use the service as normal. For more information, see Data processing pack availability change in the *AWS IoT SiteWise User Guide*.

Topics

- Versions
- Type
- Operating system
- Requirements
- Dependencies
- Configuration
- Local log file
- Licenses
- Changelog
- See also

Versions

This component has the following versions:

- 3.5.x
- 3.4.x
- 3.3.x

- 3.2.x
- 3.1.x
- 3.0.x
- 2.2.x
- 2.1.x
- 2.0.x

Type

This component is a generic component (`aws.greengrass.generic`). The Greengrass nucleus runs the component's lifecycle scripts.

For more information, see Component types.

Operating system

This component can be installed on core devices that run the following operating systems:

- Linux
- Windows

Requirements

This component has the following requirements:

- The Greengrass core device must run on one of the following platforms:
  - os: Ubuntu 20.04 or later

    architecture: x86_64 (AMD64)
  - os: Red Hat Enterprise Linux (RHEL) 8

    architecture: x86_64 (AMD64)
  - os: Amazon Linux 2

    architecture: x86_64 (AMD64)
  - os: Windows Server 2019 or later

    architecture: x86_64 (AMD64)

- os: Debian 11 (Bullseye) or later

  architecture: x86_64 (AMD64)
- The Greengrass core device must allow inbound traffic on port 443.
- The Greengrass core device must allow outbound traffic on port 443 and 8883.
- The following ports are reserved for use by AWS IoT SiteWise: 80, 443, 3001, 4569, 4572, 8000, 8081, 8082, 8084, 8085, 8086, 8445, 9000, 9500, 11080, and 50010. Using a reserved port for traffic can result in a terminated connection.

  > **Note**
  >
  > Port 8087 is required only for version 2.0.15 and later of this component.
- The Greengrass device role must have permissions that allow you to use AWS IoT SiteWise gateways on your AWS IoT Greengrass V2 devices. For more information, see Requirements in the *AWS IoT SiteWise User Guide*.

Endpoints and ports

This component must be able to perform outbound requests to the following endpoints and ports, in addition to endpoints and ports required for basic operation. For more information, see Allow device traffic through a proxy or firewall.

Endpoint	Port	Required	Description
`model.iotsitewise.region.amazonaws.com`	443	Yes	Get information about your AWS IoT SiteWise assets and asset models.

Endpoint	Port	Required	Description
edge.iotsitewise. *region*.amazonaws.com	443	Yes	Get information about the core device's AWS IoT SiteWise gateway configuration.
ecr.*region*.amazonaws.com	443	Yes	Download AWS IoT SiteWise Edge gateway Docker images from Amazon Elastic Container Registry.
iot.*region*.amazonaws.com	443	Yes	Get device endpoints for your AWS account.
sts.*region*.amazonaws.com	443	Yes	Get the ID of your AWS account.

Endpoint	Port	Required	Description
`monitor.iotsitewise. region.amazonaws.com`	443	No	Required if you access AWS IoT SiteWise Monitor portals on the core device.

Dependencies

When you deploy a component, AWS IoT Greengrass also deploys compatible versions of its dependencies. This means that you must meet the requirements for the component and all of its dependencies to successfully deploy the component. This section lists the dependencies for the released versions of this component and the semantic version constraints that define the component versions for each dependency. You can also view the dependencies for each version of the component in the AWS IoT Greengrass console. On the component details page, look for the **Dependencies** list.

The following table lists the dependencies for versions 2.0.x to 2.1.x of this component.

Dependency	Compatible versions	Dependency type
Token exchange service	>=2.0.3 <3.0.0	Hard
Stream manager	>=2.0.10 <3.0.0	Hard
Greengrass CLI	>=2.3.0 <3.0.0	Hard

For more information about component dependencies, see the component recipe reference.

Configuration

This component doesn't have any configuration parameters.

Local log file

This component uses the following log file.

Linux

```
/greengrass/v2/logs/aws.iot.SiteWiseEdgeProcessor.log
```

Windows

```
C:\greengrass\v2\logs\aws.iot.SiteWiseEdgeProcessor.log
```

To view this component's logs

- Run the following command on the core device to view this component's log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder.

  Linux

  ```
  sudo tail -f /greengrass/v2/logs/aws.iot.SiteWiseEdgeProcessor.log
  ```

  Windows (PowerShell)

  ```
  Get-Content C:\greengrass\v2\logs\aws.iot.SiteWiseEdgeProcessor.log -Tail 10 -
  Wait
  ```

Licenses

This component includes the following third-party software/licensing:

- Apache-2.0
- MIT
- BSD-2-Clause
- BSD-3-Clause
- CDDL-1.0

- CDDL-1.1
- ISC
- Zlib
- GPL-3.0-with-GCC-exception
- Public Domain
- Python-2.0
- Unicode-DFS-2015
- BSD-1-Clause
- OpenSSL
- EPL-1.0
- EPL-2.0
- GPL-2.0-with-classpath-exception
- MPL-2.0
- CC0-1.0
- JSON

This component is released under the Greengrass Core Software License Agreement.

Changelog

The following table describes the changes in each version of the component.

Version	Changes
3.5.1	New features Added support for ingestion of Null and NaN values if ingestion is enabled in AWS IoT SiteWise. To view or modify the Null and NaN configuration in AWS IoT SiteWise, see the DescribeStorageConfiguration and PutStorageConfiguration APIs. Bug fixes and improvements Updated dependencies to address potential security vulnerabilities.

Version	Changes
3.4.0	New features • Added optional HTTP and HTTPS proxy support to enable gateway communication through proxy servers that require custom certificates. For more information on setting up an HTTP proxy for the AWS IoT Greengrass core, see Connect on port 443 or through a network proxy. For more information on proxy support specific to SiteWise Edge, see Manage trust stores for AWS IoT SiteWise Edge proxy support in the *AWS IoT SiteWise User Guide*. • Added configurable session timeout settings to manage inactivity periods for AWS OpsHub and SiteWise Edge APIs. For more informati on, see Configure session timeouts for AWS IoT SiteWise Edge in the *AWS IoT SiteWise User Guide*. Performance improvements Reduced the time for incoming data to reach edge device storage from 5 seconds to less than 1 second. The latency for data uploads to AWS IoT SiteWise remains unchanged.
3.3.1	New feature • Added optional CORS support to SiteWise Edge APIs, enhancing cross-origin resource sharing capabilities. This feature improves flexibility for web applications interacting with the APIs.

Version	Changes
3.3.0	Performance improvements • Optimized cache refresh mechanism to reduce I/O usage between AWS IoT SiteWise asset syncs by only refreshing entries for new or updated assets. • Reduced memory footprint for maintaining a cache with a large number of synced asset properties. Bug fixes and improvements • Suppressed logs for ingesting individual property values when there are no ingestion errors, which reduces log noise during high ingestion rates. • Improved log readability by using human-readable formatting for certain log entries. • Added support for Java 17 and higher.

Version	Changes

3.2.1	Bug fixes and improvements
	• Fix issue where the AWS IoT SiteWise API calls do not paginate synchronously with SiteWise Edge.
	• Fix issue to not publish the `MessageRemaining.SiteWise_Edge_Stream` metric anymore.
	• Added the following CloudWatch metrics to monitor the connection with the MQTT broker.
	  • `IoTSiteWiseProcessor.IsConnectedToMqttBroker`
	  • `IoTSiteWiseProcessor.NumberOfSubscriptionsToMqttBroker`
	  • `IoTSiteWiseProcessor.NumberOfUniqueMqttTopicsReceived`
	  • `IoTSiteWiseProcessor.MqttMessageReceivedSuccessCount`
	  • `IoTSiteWiseProcessor.MqttReceivedSuccessBytes`
	  For more information about these metrics, see AWS IoT Greengrass Version 2 gateway metrics.

Version	Changes
3.2.0	Performance improvements • Optimize API services to have smaller memory footprint and require less disk space to install • This provides a 2 GB reduction in initial memory usage (now uses 7.5 GB of memory on startup, however 16 GB is still recommend ed) and 500 MB reduction in download size (now requires a 1.4 GB download) for the entire component. New features • `GetAssetPropertyValueAggregates` API now supports 15 minute aggregation windows on the edge. • Ports 8081 and 8082 no longer need to be available for this component to run correctly. **Note** The local endpoint for AWS IoT SiteWise data plane APIs, such as `get-asset-property-value` , is being changed from `http://localhost:8081` to `http://localhost:11080/data` . The local endpoint for AWS IoT SiteWise control plane APIs, such as `list-asset-models` , is being changed from `http://localhost:11080` to `http://localhost:11080/control` . AWS always recommends that you use the SiteWise Edge gateway HTTPS endpoints. Those endpoints have not changed. Bug fixes and improvements • Syncing from AWS IoT SiteWise will now transition resources into a valid state if the previous sync was interrupted. This will fix issues with some resources being corrupted after a forced restart. • Fixes a rare condition where a resource may be corrupted on the edge if it is modified during sync. Sync will now fail if this condition is detected, and the resource will be retried in the next sync.

Version	Changes
	• Fixes an issue that could have allowed the HTTP endpoint for APIs to be called externally. Only HTTPS can be used to call APIs outside of the local loopback address now. • `ListAssets` API now shows the asset hierarchies for assets stored on the edge. • Fixes an issue where the Data Processing Pack failed to restart, upgrade, or downgrade on Windows. • Fixes a bug in the Data Processing Pack for Windows OS that prevented customers from using credentials to connect with an MQTT Broker.
3.1.3	Bug fixes and improvements • Fix issue where the Data Processing Pack incorrectly reported a successful sync when some of the resources actually failed. • Allow multiple assets to have the same name as long as they don't have the same parent.
3.1.1	Bug fixes and improvements • Fix issue where SigV4 request fails due to a timezone mismatch. • Fix issue where transform and metric properties stop calculating when they rely on attributes after restarting. • Enable support of custom Stream Manager Port configuration. • Fix an issue where properties that are synced to the edge might stop getting updated.
3.1.0	Bug fixes and improvements • Fix issue where `ListAssetModels` API fails to generate next token.
3.0.0	New features • Enables support of data ingestion from an MQTT broker.

Version	Changes
2.2.1	Bug fixes and improvements • Adjust the sync process in order to make control plane data storage more consistent with how cloud operates. This slightly impacts upgrading. **Note** Control plane data synced on version 2.2.1 or higher won't be compatible with previous versions. To downgrade to previous versions, you'll need to complete a fresh install. This doesn't impact upgrades, data synced on previous versions will work with version 2.2.1. • Additional modifications to the AWS credentials chain to prioritize AWS IoT Greengrass V2 credentials.
2.1.37	Bug fixes and improvements • Deprecate dependency-routing-service process and move its functiona lity into the property-state-service process to reduce resource usage from the processes communicating. • Increase maximum result limit for the `get-asset-property-value-history` API to 20,000 to match the limit used by AWS IoT SiteWise. • Fix an issue where next token wasn't being provided in paginated results for the `get-asset-property-value-history` API when no max result limit was specified.
2.1.35	Bug fixes and improvements • Modifies the AWS credentials chain to prioritize AWS IoT Greengrass credentials. • Fixes an issue with account detection when deploying as part of an AWS IoT Thing group.

Version	Changes
2.1.34	Bug fixes and improvements • Adjusts metric/transform computations to use multi-threading on Linux. Windows continues to run single-threaded computations for compatibility. • Fixes an issue where metric computations would be missing for some computation windows.
2.1.33	Bug fixes and improvements • Fixes an issue with error state reporting to the Greengrass console.
2.1.32	Bug fixes and improvements • Adds support for customized user names and groups.
2.1.31	Bug fixes and improvements • Adds support to compute the time-weighted average and the time-weighted standard deviation for data that is modeled in AWS IoT SiteWise.
2.1.29	Bug fixes and improvements • Adds support to filter assets on the edge functionality.
2.1.28	Bug fixes and improvements • Optimizes resource synchronization to enable a large number of assets to sync from the AWS Cloud to the edge.
2.1.24	Bug fixes and improvements • Fixes an issue that caused the dashboard to disappear when syncing a resource for the second time.

Version	Changes
2.1.23	Bug fixes and improvements • Added a timeout for the `aws.iot.SiteWiseEdgeProcessor` install process to avoid installation failure if internet connectivity is slow. • Optimized resource sync to improve sync efficiency between the cloud and edge.
2.1.21	**Warning** Upgrading from 2.0.x to 2.1.x will result in loss of local data. New features • Adds support for Windows Server 2019 or higher. • Removes docker for Linux-based operating systems.
2.0.16	This version contains bug fixes and improvements.
2.0.15	Bug fixes and improvements • Changes the port that this component uses for resource sync API operations from 8085 to 8087. As a result, this component now requires port 8087 to be available. This component still requires port 8085 to be available. • Updates AWS OpsHub authentication to deny unauthorized users during login instead of when a user attempts to call API operations.
2.0.14	This version contains bug fixes and improvements.
2.0.13	Bug fixes and improvements • Fixes an issue so that when this component reports data to Amazon CloudWatch metrics, it now correctly indicates which data is unmodeled.

Version	Changes
2.0.9	Bug fixes and improvements • Improves reliability to create and update AWS IoT SiteWise resources on the core device. • Adds additional local API operations that you can use to monitor which components are installed on the core device, the version of each component, and the status of each component. You can view this information on the **Settings** tab in the AWS OpsHub for AWS IoT SiteWise application on the core device. • Adds a health status for the Docker containers that this component runs. You can run the `docker ps` command to view the containers' health status.
2.0.7	Bug fixes and improvements • Fixes support for viewing AWS IoT SiteWise Monitor portals on the core device.
2.0.6	Bug fixes and improvements • Fixes the AWS IoT SiteWise `statetime()` , `earliest()` , and `latest()` functions that this component computes on the core device.
2.0.5	Bug fixes and improvements • Adds support for the AWS IoT SiteWise `pretrigger()` function in transforms that this component computes on the core device. • Changes the path where this component stores the Lightweight Directory Access Protocol (LDAP) configuration for authentication.
2.0.2	Initial version.

See also

- What is AWS IoT SiteWise? in the *AWS IoT SiteWise User Guide*.

Publisher-supported components

Publisher-supported components are in a preview release for AWS IoT Greengrass and are subject to change. These components are not supported by AWS. You must contact the Publisher for any issues with each of the components.

The Greengrass Publisher-supported components are developed, offered, and serviced by third-party component vendors. Third-party component vendors are either from the AWS Partner Device Catalog, AWS Heroes, or community vendors. You can purchase the components in this catalog by contacting the third-party component vendor directly.

The Greengrass Publisher-supported components include the following:

Topics

- AIShield.Edge
- AI EdgeLabs Sensor
- Greengrass S3 Ingestor

AIShield.Edge

This component was developed and is supported by AIShield, powered by Bosch. Boost your AI security with AIShield.Edge. This component is designed to seamlessly deploy threat-informed, tailored defenses to edge devices, which safeguardes your devices against AI attacks.

This component offers the following benefits:

- Seamlessly transition from vulnerability analysis with AIShield AI Security to fortified edge defenses within AWS
- Deploy tailored defenses across multiple edge devices with ease
- Broad protection tailored to diverse AI setups which supports various model types and frameworks
- Stay updated with seamless integration into Amazon SageMaker AI and Greengrass workflows
- Gain immediate insights into potential threats, with data relayed directly to AWS IoT Core
- A cohesive AI security pathway for defense deployment on the edge from AIShield AI Security on the AWS Marketplace

This component must run on the following platform:

- os: linux

If you are interested in purchasing this component, contact Bosch Software and Digital Solutions: <AIShield.Contact@bosch.com>.

AI EdgeLabs Sensor

This component was developed and is supported by AI EdgeLabs. AI EdgeLabs Sensor is a container-based application that contains AI-based threat detection and prevention capabilities. AI Sensor is wrapped into a Greengrass component and deployed as a standalone container on the core device alongside other Greengrass components.

This current component is a container-based agent that continuously verifies network communication, looks for threat patterns in software running on the Edge Host or IoT gateway. This component uses eBPF, behavioral verification of processes bandwidth, and the host-based configuration. The main functionality of this component is based on NDR/IPS and EDR functions.

This component offers the following benefits:

- AI-based threat detection against network attacks and malware (EDR/NDR)
- Automated AI-based Incident response (IPS)
- Host-local threat intelligence with minimal data-transfer outside
- Lightweight deployment with Docker and Greengrass

This component must run on one of the following platforms:

- os: linux

If you are interested in purchasing this component, contact AI EdgeLabs: <contact@edgelabs.ai>.

Greengrass S3 Ingestor

This component was developed and is supported by Nathan Glover. The Greengrass S3 Ingestor component is designed to be used with the stream manager component. This component takes a

line-delimited stream of JSON messages from stream manager and batches them into a GZIP file. This component enables efficient ingestion of data into Amazon S3 for further processing or for storage. This component doesn't support sending data to the AWS Cloud in realtime.

This component must run on one of the following platforms:

- os: linux
- os: Windows

If you are interested in purchasing this component, contact Nathan Glover: `<nathan@glovers.id.au>`.

Community components

The Greengrass Software Catalog is an index of Greengrass components that are developed by the Greengrass community. From this catalog, you can download, modify, and deploy components to create your Greengrass applications. You can view the catalog at the following link: https://github.com/aws-greengrass/aws-greengrass-software-catalog.

Each component has a public GitHub repository that you can explore. View the Greengrass Software Catalog on GitHub to find the full list of community components. For example, this catalog includes the following components:

- **Amazon Kinesis Video Streams**

  This component ingests audio and video streams from local cameras that use Real Time Streaming Protocol (RTSP). The component then uploads the audio and video streams to Amazon Kinesis Video Streams.

- **Bluetooth IoT gateway**

  This component uses the BluePy library that enables communication with Bluetooth Low Energy (LE) devices to create Bluetooth LE client interfaces.

- **Certificate Rotator**

  This component provides a means of rotating the AWS IoT Greengrass core device certificate and private key, across your fleet, at scale.

- **Containerized secure tunneling**

This component provides a Docker container for secure tunneling with all dependencies and matching libraries in a reusable recipe that doesn't rely on a specific host operating system.

- **Grafana**

  This component enables you to host a Grafana server on a Greengrass core device. You can use Grafana dashboards to visualize and manage data on the core device.

- **GStreamer for Amazon Lookout for Vision**

  This component provides a GStreamer plugin so that you can perform Lookout for Vision anomaly detection in your custom GStreamer pipelines.

- **Home assistant**

  This component enables the customer to use Home Assistant to provide local control of smart home devices. It provides integration with AWS services at the edge and in the cloud to deliver home automation solutions that extend Home Assistant.

- **InfluxDBGrafana dashboard**

  This component provides a one-click experience to set up the InfluxDB and Grafana components. It connects InfluxDB to Grafana and automates the setup of a local Grafana dashboard that renders AWS IoT Greengrass telemetry in real time.

- **InfluxDB**

  This component provides an InfluxDB time-series database on a Greengrass core device. You can use this component to process data from IoT sensors, analyze data in real time, and monitor operations at the edge.

- **InfluxDB publisher**

  This component relays AWS IoT Greengrass system health telemetry from the Nucleus emitter plugin to InfluxDB. This component can also forward custom telemetry to InfluxDB.

- **IoT pubsub framework**

  This framework provides an application architecture, template code, and deployable examples that help improve code quality for distributed event-driven IoT pubsub applications using AWS IoT Greengrass v2 custom components. For more information, see Create AWS IoT Greengrass components.

- **Jupyter Labs**

This component deploys JupyterLab to an AWS IoT Greengrass core device. The Jupyter environment has access to the process and environment variable resources set by AWS IoT Greengrass, simplifying the process of testing and developing components written in Python.

- **Local web server**

  This component enables you to create a local web user interface on a Greengrass core device. You can create a local web user interface that enables you to configure device and application settings or monitor the device, for example.

- **LoRaWaN protocol adapter**

  This component ingests data from local wireless devices that use the LoRaWaN protocol, which is a low-power wide-area network (LPWAN) protocol. The component enables you to analyze and act on data locally without communicating with the cloud.

- **Modbus TCP**

  This component collects data from local devices using the ModbusTCP protocol and publishes it to selected data streams.

- **Node-RED**

  This component installs Node-RED on an AWS IoT Greengrass core device using NPM. The component depends on the Node-RED Auth component which must be explicitly deployed and configured. You can use the Node-RED CLI for Greengrass to deploy Node-RED flows to AWS IoT Greengrass devices.

- **Node-RED Docker**

  This component installs Node-RED on the AWS IoT Greengrass core device using the official Node-RED Docker container. The component depends on the Node-RED Auth component which must be explicitly deployed and configured. You can use the Node-RED CLI for Greengrass to deploy Node-RED flows to AWS IoT Greengrass devices.

- **Node-RED Auth**

  This component configures a user name and password to secure the Node-RED instance running on an AWS IoT Greengrass core device.

- **OpenThread Border Router**

  This component deploys the OpenThread Border Router Docker container. The component helps to compose a Matter device that includes a Thread border router.

- **OSI Pi Streaming Data Connector**

  This component provides streaming real-time data ingestion from OSI Pi Data Archive to a modern data architecture on AWS. It integrates to OSI Pi Asset Framework that is centrally managed over AWS IoT PubSub messaging.

- **Parsec Provider**

  This component enables AWS IoT Greengrass devices to integrate hardware security solutions using the open source Parsec project from Cloud Native Computing Foundation (CNCF).

- **PostgreSQL DB**

  This component provides support for PostgreSQL relational database at the edge. Customers can use this component to provision and manage a local PostgreSQL instance inside a docker container.

- **S3 file uploader**

  This component monitors a directory for new files, uploads them to Amazon Simple Storage Service (Amazon S3), and then deletes them after a successful upload.

- **Secrets Manager client**

  This component provides a CLI tool that can be used by other components needing to retrieve secrets from the Secrets Manager component in a recipe lifecycle script.

- **TES routing to container**

  This component configures nftables or iptables on an AWS IoT Greengrass device so that it can use the Token exchange service component with containers.

- **WebRTC**

  This component ingests audio and video streams from RTSP cameras connected to the AWS IoT Greengrass core device. And then the component turns the audio and video streams into peer-to-peer communication or relay through Amazon Kinesis Video Streams.

To request a feature or report a bug, open a GitHub issue on the repository for that component. AWS doesn't provide support for community components. For more information, see the **CONTRIBUTING.md** file in each component's repository.

Several AWS-provided components are also open source. For more information, see Open source AWS IoT Greengrass Core software.

AWS IoT Greengrass development tools

Use AWS IoT Greengrass development tools to create, test, build, publish, and deploy custom Greengrass components.

- **Greengrass Development Kit CLI**

  Use the AWS IoT Greengrass Development Kit Command-Line Interface (GDK CLI) in your local development environment to create components from templates and community components in the Greengrass Software Catalog. You can use the GDK CLI to build the component and publish the component to the AWS IoT Greengrass service as a private component in your AWS account.

- **Greengrass Command Line Interface**

  Use the Greengrass Command Line Interface (Greengrass CLI) on Greengrass core devices to deploy and debug Greengrass components. The Greengrass CLI is a component that you can deploy to your core devices to create local deployments, view details about installed components, and explore log files.

- **Local debug console**

  Use the local debug console on Greengrass core devices to deploy and debug Greengrass components using a local dashboard web interface. The local debug console is a component that you can deploy to your core devices to create local deployments and view details about installed components.

AWS IoT Greengrass also provides the following SDKs that you can use in custom Greengrass components:

- The AWS IoT Device SDK and the AWS IoT Greengrass Component SDK, which contain the interprocess communication (IPC) library. For more information, see Use the AWS IoT Device SDK to communicate with the Greengrass nucleus, other components, and AWS IoT Core.
- The Stream Manager SDK, which you can use to transfer data streams to the AWS Cloud. For more information, see Manage data streams on Greengrass core devices.

Topics

- AWS IoT Greengrass Development Kit Command-Line Interface
- Greengrass Command Line Interface

- Use AWS IoT Greengrass Testing Framework

AWS IoT Greengrass Development Kit Command-Line Interface

The AWS IoT Greengrass Development Kit Command-Line Interface (GDK CLI) provides features that help you develop custom Greengrass components. You can use the GDK CLI to create, build, and publish custom components. When you create a component repository with the GDK CLI, you can start from a template or a community component from the Greengrass Software Catalog. Then, you can choose a build system that packages files as ZIP archives, uses a Maven or Gradle build script, or runs a custom build command. After you create a component, you can use the GDK CLI to publish it to the AWS IoT Greengrass service, so you can use the AWS IoT Greengrass console or API to deploy the component to your Greengrass core devices.

When you develop Greengrass components without the GDK CLI, you must update the version and artifact URIs in the component recipe file each time you create a new version of the component. When you use the GDK CLI, it can automatically update the version and artifact URIs for you each time you publish a new version of the component.

The GDK CLI is open source and available on GitHub. You can customize and extend the GDK CLI to meet your component development needs. We invite you to open issues and pull requests on the GitHub repository. You can find the GDK CLI source at the following link: https://github.com/aws-greengrass/aws-greengrass-gdk-cli.

Prerequisites

To install and use the Greengrass Development Kit CLI, you need the following:

- An AWS account. If you don't have one, see Set up an AWS account.
- A Windows, macOS, or Unix-like development computer with an internet connection.
- For GDK CLI version 1.1.0 or later, Python 3.6 or later installed on your development computer.

  For GDK CLI version 1.0.0, Python 3.8 or later installed on your development computer.
- Git installed on your development computer.
- AWS Command Line Interface (AWS CLI) installed and configured with credentials on your development computer. For more information, see Installing, updating, and uninstalling the AWS CLI and Configuring the AWS CLI in the *AWS Command Line Interface User Guide*.

> **Note**
>
> If you use a Raspberry Pi or another 32-bit ARM device, install AWS CLI V1. AWS CLI V2 isn't available for 32-bit ARM devices. For more information, see Installing, updating, and uninstalling the AWS CLI version 1.

- To use the GDK CLI to publish components to the AWS IoT Greengrass service, you must have the following permissions:
  - `s3:CreateBucket`
  - `s3:GetBucketLocation`
  - `s3:PutObject`
  - `greengrass:CreateComponentVersion`
  - `greengrass:ListComponentVersions`
- To use the GDK CLI to build a component whose artifacts exist in an S3 bucket and not in the local file system, you must have the following permissions:
  - `s3:ListBucket`

  This feature is available for GDK CLI v1.1.0 and later.

Changelog

The following table describes the changes in each version of the GDK CLI. For more information, see the GDK CLI Releases page on GitHub.

Version	Changes
1.6.2	Bug fixes and improvements • Fixes an issue where Windows gradlew.bat does not work due to the relative path. • Minor improvements to logging, testing, and packaging.
1.6.1	Bug fixes and improvements • Adds a security fix for CLI argument parsing.

Version	Changes
	• Enables the GDK to get the latest Greengrass Testing Framework (GTF) release name as the default GTF version. • Enables GDK to recommend customers using an older version of GTF that they update to the latest version.
1.6.0	New features • Adds a recipe validation check against the Greengrass recipe schema during the `component build` and `component publish` commands. This update helps developers to identify actionable issues within their component recipes earlier in the component creation process. • Adds a confidence test suite to the template that can be pulled down by the `test-e2e init` command. This confidence test suite includes eight generic tests that can be used and extended to fit basic component testing needs. Bug fixes and improvements • Updates the default Greengrass Testing Framework (GTF) version used by the `test-e2e` command to version 1.2.0.
1.5.0	Bug fixes and improvements Updates the patterns recognized by the `excludes` build option when `build_system` is `zip`. This version will now recognize glob patterns which match pathnames based on their wildcard characters. This enables custom specification of which directories to exclude from.

Version	Changes
1.4.0	New features • Adds a new `config` command that starts an interactive prompt to modify fields within an existing GDK configuration file. • Modifies the `gdk component build` and `gdk component publish` commands to verify that the recipe size is within Greengrass requirements (<=16000 bytes) before proceeding. Bug fixes and improvements • Adds additional logging in the output of the `gdk component build` command when a recipe syntax error is preventing the build from completing for awareness. • Renames the `otf-options` and `otf-version` to `gtf-options` and `gtf-version` respectively, due to the renaming of Open Test Framework to Greengrass Testing Framework.
1.3.0	New features • Adds a new `test-e2e` command to support end-to-end testing of components using Open Test Framework. • Adds a new configuration option, `zip_name`, to support configurable zip file names with the zip build system. • Makes the `region` property in the GDK configuration file optional. Bug fixes and improvements • Fixes an issue where a new directory is created even when the specified template or repository doesn't exist when initializing a GDK project with the `--name` argument.
1.2.3	Bug fixes and improvements • Fixes an issues where bucket creation fails due to incorrect error handling. • Fixes an issue where list structures in the component recipe are removed.

Version	Changes
1.2.2	Bug fixes and improvements • Recipe keys are no longer case sensitive. • Adds a check to determine if a bucket exists in an AWS Region and is accessible by the user before creating a new bucket. Requires the user to have the `GetBucketLocation` permission. • Fixes an issue with the `excludes` keyword in the GDK CLI configura tion file.
1.2.1	Bug fixes and improvements • Accepts the Canada (Central) (`ca-central-1`) AWS Region in the region configuration entry in the `gdk-config.json` file. • Fixes issues with the `--region` GDK CLI argument to the `publish` command.
1.2.0	New features • Adds the `options` entry to the `build` configuration in the GDK CLI configuration file. Supports `excludes` under `options` to exclude certain files from the zip artifact when using the `zip` build system. • Adds the `gradlew` build system to use Gradle Wrapper to build components. • Adds support for Kotlin DSL build files for the `gradle` build option. • Adds an `options` entry to the `publish` configuration in the GDK CLI configuration file. Supports the `file_upload_args` under `options` to provide extra arguments when uploading files to Amazon S3. Bug fixes and improvements • Fixes an issue where Gradle builds didn't clean before running a build command. • Fixes an issue where the build didn't exit when the build command fails. • Improves the output format of the `gdk component list` command.

Version	Changes
1.1.0	New features • Adds support for the Gradle build system. • Adds support for the Maven build system on Windows devices. • Adds the `--bucket` argument to the component publish command. You can use this argument to specify the exact bucket where the GDK CLI uploads component artifacts. • Adds the `--name` argument to the component init command. You can use this option to specify the folder where the GDK CLI initializes the component. • Adds support for component artifacts that exist in an S3 bucket but not in the local component build folder. You can use this feature to reduce bandwidth costs for large component artifacts, such as machine learning models. Bug fixes and improvements • Updates the component publish command to check if the component is built before it publishes the component. If the component isn't built, this command now builds the component for you. • Fixes an issue where the zip build system fails to build on Windows devices when the ZIP file name contains capital letters. • Improves the log message format and changes the default log level to `INFO` on devices that run Python versions earlier than 3.8. • Changes the minimum Python version requirement to Python 3.6.
1.0.0	Initial version.

Install or update the AWS IoT Greengrass Development Kit Command-Line Interface

The AWS IoT Greengrass Development Kit Command-Line Interface (GDK CLI) is built on Python, so you can use `pip` to install it on your development computer.

> **Tip**
>
> You can also install the GDK CLI in a Python virtual environments such as venv. For more information, see Virtual Environments and Packages in the *Python 3 documentation*.

To install or update the GDK CLI

1. Run the following command to install the latest version of the GDK CLI from its GitHub repository.

   ```
   python3 -m pip install -U git+https://github.com/aws-greengrass/aws-greengrass-gdk-cli.git@v1.6.2
   ```

   > **Note**
   >
   > To install a specific version of the GDK CLI, replace *versionTag* with the version tag to install. You can view version tags for the GDK CLI in its GitHub repository.
   >
   > ```
   > python3 -m pip install -U git+https://github.com/aws-greengrass/aws-greengrass-gdk-cli.git@versionTag
   > ```

2. Run the following command to verify that the GDK CLI installed successfully.

   ```
   gdk --help
   ```

   If the gdk command isn't found, add its folder to PATH.

   - On Linux devices, add /home/*MyUser*/.local/bin to PATH, and replace *MyUser* with the name of your user.
   - On Windows devices, add *PythonPath*\\Scripts to PATH, and replace *PythonPath* with the path to the Python folder on your device.

You can now use the GDK CLI to create, build, and publish Greengrass components. For more information about how to use the GDK CLI, see AWS IoT Greengrass Development Kit Command-Line Interface commands.

AWS IoT Greengrass Development Kit Command-Line Interface commands

The AWS IoT Greengrass Development Kit Command-Line Interface (GDK CLI) provides a command line interface that you can use to create, build, and publish Greengrass components on your development computer. GDK CLI commands use the following format.

```
gdk <command> <subcommand> [arguments]
```

When you install the GDK CLI, the installer adds gdk to the PATH so you can run the GDK CLI from the command line.

You can use the following arguments with any command:

- Use `-h` or `--help` for information about a GDK CLI command.
- Use `-v` or `--version` to see what version of GDK CLI is installed.
- Use `-d` or `--debug` to output verbose logs that you can use to debug the GDK CLI.

This section describes the GDK CLI commands and provides examples for each command. The synopsis for each command shows its arguments and their usage. Optional arguments are shown in square brackets.

Available commands

- component
- config
- test-e2e

component

Use the `component` command in the AWS IoT Greengrass Development Kit Command-Line Interface (GDK CLI) to create, build, and publish custom Greengrass components.

Subcommands

- init
- build
- publish
- list

init

Initialize a Greengrass component folder from a component template or community component.

The GDK CLI retrieves community components from the Greengrass Software Catalog and component templates from the AWS IoT Greengrass Component Templates repository on GitHub.

> **Note**
>
> If you use GDK CLI v1.0.0, you must run this command in an empty folder. The GDK CLI downloads the template or community component to the current folder.
> If you use GDK CLI v1.1.0 or later, you can specify the `--name` argument to specify the folder where the GDK CLI downloads the template or community component. If you use this argument, specify a folder that doesn't exist. The GDK CLI creates the folder for you. If you don't specify this argument, the GDK CLI uses the current folder, which must be empty.
> If the component uses the zip build system, the GDK CLI zips certain files in the component's folder into a zip file with the same name as the component folder. For example, if the component folder's name is `HelloWorld`, the GDK CLI creates a zip file named `HelloWorld.zip`. In the component recipe, the zip artifact name must match the name of the component folder. If you use GDK CLI version 1.0.0 on a Windows device, the component folder and zip file names must contain only lowercase letters.
> If you initialize a template or community component that uses the zip build system to a folder with a different name than the template or component, you must change the zip artifact name in the component recipe. Update the `Artifacts` and `Lifecycle` definitions such that the zip file name matches the name of the component folder. The following example highlights the zip file name in the `Artifacts` and `Lifecycle` definitions.
>
> JSON
>
> ```
> {
>   ...
>   "Manifests": [
>     {
>       "Platform": {
>         "os": "all"
>       },
>       "Artifacts": [
>         {
>           "URI": "s3://BUCKET_NAME/COMPONENT_NAME/
> COMPONENT_VERSION/HelloWorld.zip",
> ```

```
          "Unarchive": "ZIP"
        }
      ],
      "Lifecycle": {
        "Run": "python3 -u {artifacts:decompressedPath}/HelloWorld/main.py
 {configuration:/Message}"
      }
    }
  ]
}
```

YAML

```
---
...
Manifests:
  - Platform:
      os: all
    Artifacts:
      - URI: "s3://BUCKET_NAME/COMPONENT_NAME/
COMPONENT_VERSION/HelloWorld.zip"
        Unarchive: ZIP
    Lifecycle:
      Run: "python3 -u {artifacts:decompressedPath}/HelloWorld/main.py
 {configuration:/Message}"
```

Synopsis

```
$ gdk component init
    [--language]
    [--template]
    [--repository]
    [--name]
```

Arguments (initialize from component template)

- `-l`, `--language` – The programming language to use for the template that you specify.

  You must specify either `--repository` or `--language` and `--template`.
- `-t`, `--template` – The component template to use for a local component project. To view available templates, use the list command.

You must specify either `--repository` or `--language` and `--template`.

- `-n`, `--name` – (Optional) The name of the local folder where the GDK CLI initializes the component. Specify a folder that doesn't exist. The GDK CLI creates the folder for you.

  This feature is available for GDK CLI v1.1.0 and later.

Arguments (initialize from community component)

- `-r`, `--repository` – The community component to check out into the local folder. To view available community components, use the list command.

  You must specify either `--repository` or `--language` and `--template`.

- `-n`, `--name` – (Optional) The name of the local folder where the GDK CLI initializes the component. Specify a folder that doesn't exist. The GDK CLI creates the folder for you.

  This feature is available for GDK CLI v1.1.0 and later.

Output

The following example shows the output produced when you run this command to initialize a component folder from the Python Hello World template.

```
$ gdk component init -l python -t HelloWorld
[2021-11-29 12:51:40] INFO - Initializing the project directory with a python
 component template - 'HelloWorld'.
[2021-11-29 12:51:40] INFO - Fetching the component template 'HelloWorld-python'
 from Greengrass Software Catalog.
```

The following example shows the output produced when you run this command to initialize a component folder from a community component.

```
$ gdk component init -r aws-greengrass-labs-database-influxdb
[2022-01-24 15:44:33] INFO - Initializing the project directory with a component
 from repository catalog - 'aws-greengrass-labs-database-influxdb'.
[2022-01-24 15:44:33] INFO - Fetching the component repository 'aws-greengrass-labs-
database-influxdb' from Greengrass Software Catalog.
```

build

Build a component's source into a recipe and artifacts that you can publish to the AWS IoT Greengrass service. The GDK CLI runs the build system that you specify in the GDK CLI

configuration file, `gdk-config.json`. You must run this command in the same folder where the `gdk-config.json` file exists.

When you run this command, the GDK CLI creates a recipe and artifacts in the `greengrass-build` folder in the component folder. The GDK CLI saves the recipe in the `greengrass-build/recipes` folder and saves the artifacts in the `greengrass-build/artifacts/componentName/componentVersion` folder.

If you use GDK CLI v1.1.0 or later, the component recipe can specify artifacts that exist in an S3 bucket but not in the local component build folder. You can use this feature to reduce bandwidth usage when you develop components with large artifacts, such as machine learning models.

After you build a component, you can do one of the following to test it on a Greengrass core device:

- If you develop on a different device than where you run the AWS IoT Greengrass Core software, you must publish the component to deploy it to a Greengrass core device. Publish the component to the AWS IoT Greengrass service, and deploy it to the Greengrass core device. For more information, see the publish command and Create deployments.
- If you develop on the same device where you run the AWS IoT Greengrass Core software, you can publish the component to the AWS IoT Greengrass service to deploy, or you can create a local deployment to install and run the component. To create a local deployment, use the Greengrass CLI. For more information, see Greengrass Command Line Interface and Test AWS IoT Greengrass components with local deployments. When you create the local deployment, specify `greengrass-build/recipes` as the recipes folder and `greengrass-build/artifacts` as the artifacts folder.

Synopsis

```
$ gdk component build
```

Arguments

None

Output

The following example shows the output produced when you run this command.

```
$ gdk component build
```

```
[2021-11-29 13:18:49] INFO - Getting project configuration from gdk-config.json
[2021-11-29 13:18:49] INFO - Found component recipe file 'recipe.yaml' in the
 project directory.
[2021-11-29 13:18:49] INFO - Building the component 'com.example.PythonHelloWorld'
 with the given project configuration.
[2021-11-29 13:18:49] INFO - Using 'zip' build system to build the component.
[2021-11-29 13:18:49] WARNING - This component is identified as using 'zip' build
 system. If this is incorrect, please exit and specify custom build command in the
 'gdk-config.json'.
[2021-11-29 13:18:49] INFO - Zipping source code files of the component.
[2021-11-29 13:18:49] INFO - Copying over the build artifacts to the greengrass
 component artifacts build folder.
[2021-11-29 13:18:49] INFO - Updating artifact URIs in the recipe.
[2021-11-29 13:18:49] INFO - Creating component recipe in 'C:\Users\MyUser\Documents
\greengrass-components\python\HelloWorld\greengrass-build\recipes'.
```

publish

Publish this component to the AWS IoT Greengrass service. This command uploads build artifacts to an S3 bucket, updates the artifact URI in the recipe, and creates a new version of component from the recipe. The GDK CLI uses the S3 bucket and AWS Region that you specify in the GDK CLI configuration file, `gdk-config.json`. You must run this command in the same folder where the `gdk-config.json` file exists.

If you use GDK CLI v1.1.0 or later, you can specify the `--bucket` argument to specify the S3 bucket where the GDK CLI uploads the component's artifacts. If you don't specify this argument, the GDK CLI uploads to the S3 bucket whose name is *`bucket`*`-`*`region`*`-`*`accountId`*, where *`bucket`* and *`region`* are the values that you specify in `gdk-config.json`, and *`accountId`* is your AWS account ID. The GDK CLI creates the bucket if it doesn't exist.

If you use GDK CLI v1.2.0 or later, You can override the AWS Region specified in the GDK CLI configuration file using the `--region` parameter. You can also specify additional options using the `--options` parameter. For a list of available options, see Greengrass Development Kit CLI configuration file.

When you run this command, the GDK CLI publishes the component with the version that you specify in the recipe. If you specify `NEXT_PATCH`, the GDK CLI uses the next patch version that doesn't already exist. Semantic versions use a *major.minor.patch* numbering system. For more information, see the semantic version specification.

Note

If you use GDK CLI v1.1.0 or later, when you run this command, the GDK CLI checks if the component is built. If the component isn't built, the GDK CLI builds the component before it publishes the component.

Synopsis

```
$ gdk component publish
    [--bucket] [--region] [--options]
```

Arguments

- `-b`, `--bucket` – (Optional) Specify the name of the S3 bucket where the GDK CLI publishes component artifacts.

  If you don't specify this argument, the GDK CLI uploads to the S3 bucket whose name is *`bucket`*`-`*`region`*`-`*`accountId`*, where *`bucket`* and *`region`* are the values that you specify in `gdk-config.json`, and *`accountId`* is your AWS account ID. The GDK CLI creates the bucket if it doesn't exist.

  The GDK CLI creates the bucket if it doesn't exist.

  This feature is available for GDK CLI v1.1.0 and later.
- `-r`, `--region` – (Optional) Specify the name of the AWS Region to when the component is created. This argument overrides the Region name in the GDK CLI configuration.

  This feature is available for GDK CLI v1.2.0 and later.
- `-o`, `--options` (Optional) Specify a list of options for publishing a component. The argument must be a valid JSON string or a file path to a JSON file containing the publishing options. This argument overrides the options in the GDK CLI configuration.

  This feature is available for GDK CLI v1.2.0 and later.

Output

The following example shows the output produced when you run this command.

```
$ gdk component publish
[2021-11-29 13:45:29] INFO - Getting project configuration from gdk-config.json
```

```
[2021-11-29 13:45:29] INFO - Found component recipe file 'recipe.yaml' in the
 project directory.
[2021-11-29 13:45:29] INFO - Found credentials in shared credentials file: ~/.aws/
credentials
[2021-11-29 13:45:30] INFO - Publishing the component 'com.example.PythonHelloWorld'
 with the given project configuration.
[2021-11-29 13:45:30] INFO - No private version of the component
 'com.example.PythonHelloWorld' exist in the account. Using '1.0.0' as the next
 version to create.
[2021-11-29 13:45:30] INFO - Uploading the component built artifacts to s3 bucket.
[2021-11-29 13:45:30] INFO - Uploading component artifacts to S3 bucket: {bucket}.
 If this is your first time using this bucket, add the 's3:GetObject' permission
 to each core device's token exchange role to allow it to download the component
 artifacts. For more information, see https://docs.aws.amazon.com/greengrass/v2/
developerguide/device-service-role.html.
[2021-11-29 13:45:30] INFO - Not creating an artifacts bucket as it already exists.
[2021-11-29 13:45:30] INFO - Updating the component recipe
 com.example.PythonHelloWorld-1.0.0.
[2021-11-29 13:45:30] INFO - Creating a new greengrass component
 com.example.PythonHelloWorld-1.0.0
[2021-11-29 13:45:30] INFO - Created private version '1.0.0' of the component in the
 account.'com.example.PythonHelloWorld'.
```

list

Retrieve the list of available component templates and community components.

The GDK CLI retrieves community components from the Greengrass Software Catalog and component templates from the AWS IoT Greengrass Component Templates repository on GitHub.

You can pass the output of this command to the init command to initialize component repositories from templates and community components.

Synopsis

```
$ gdk component list
    [--template]
    [--repository]
```

Arguments

- `-t`, `--template` – (Optional) Specify this argument to list available component templates. This command outputs the name and language of each template in the format

name-*language*. For example, in `HelloWorld-python`, the template name is `HelloWorld` and the language is `python`.

- `-r`, `--repository` – (Optional) Specify this argument to list available community component repositories.

Output

The following example shows the output produced when you run this command.

```
$ gdk component list --template
[2021-11-29 12:29:04] INFO - Listing all the available component templates from
 Greengrass Software Catalog.
[2021-11-29 12:29:04] INFO - Found '2' component templates to display.
1. HelloWorld-python
2. HelloWorld-java
```

config

Use the `config` command in the AWS IoT Greengrass Development Kit Command-Line Interface (GDK CLI) to modify the configuration for the GDK in the configuration file, `gdk-config.json`.

Subcommands

- update

update

Start an interactive prompt to modify fields within an existing GDK configuration file.

Synopsis

```
$ gdk config update
    [--component]
```

Arguments

- `-c`, `--component` – To update component-related fields in the `gdk-config.json` file. This argument is required since it is the only option.

Output

The following example shows the output produced when you run this command to configure a component.

```
$ gdk config update --component
Current value of the REQUIRED component_name is (default:
 com.example.PythonHelloWorld):
Current value of the REQUIRED author is (default: author):
Current value of the REQUIRED version is (default: NEXT_PATCH):
Do you want to change the build configurations? (y/n)
Do you want to change the publish configurations? (y/n)
[2023-09-26 10:19:48] INFO - Config file has been updated. Exiting...
```

test-e2e

Use the `test-e2e` command in the AWS IoT Greengrass Development Kit Command-Line Interface (GDK CLI) to initialize, build, and run end-to-end test modules in the GDK project.

Subcommands

- init
- build
- run

init

Initialize an existing GDK CLI project with a testing module that uses Greengrass Testing Framework (GTF).

By default, GDK CLI retrieves the maven module template from the AWS IoT Greengrass Component Templates repository on GitHub. This maven module comes with a dependency on the `aws-greengrass-testing-standalone` JAR file.

This command creates a new directory called `gg-e2e-tests` inside of the GDK project. If the testing module directory already exists and is not empty, the command exits without doing anything. This `gg-e2e-tests` folder contains the Cucumber feature and step definitions structured in a maven project.

By default, this command will try to use the latest release version of GTF.

Synopsis

```
$ gdk test-e2e init
    [--gtf-version]
```

Arguments

- `-ov`, `--gtf-version` – (Optional) The version of the GTF to use with the end-to-end testing module in the GDK project. This value must be one of the GTF versions from releases. This argument overrides the `gtf_version` in the GDK CLI configuration.

Output

The following example shows the output produced when you run this command to initialize the GDK project with the testing module.

```
$ gdk test-e2e init
[2023-12-06 12:20:28] INFO - Using the GTF version provided in the GDK test config
 1.2.0
[2023-12-06 12:20:28] INFO - Downloading the E2E testing template from GitHub into
 gg-e2e-tests directory...
```

build

> **Note**
>
> You must build the component by running **gdk component build** before building the end-to-end test module.

Build the end-to-end testing module. The GDK CLI builds the testing module using the build system that you specify in the GDK CLI configuration file, `gdk-config.json`, under the `test-e2e` property. You must run this command in the same folder where the `gdk-config.json` file exists.

By default, GDK CLI uses maven build system to build the testing module. Maven is required to run the `gdk test-e2e build` command.

You must build the component by running **gdk-component-build** before building the testing module, if the test feature files have variables like `GDK_COMPONENT_NAME` and `GDK_COMPONENT_RECIPE_FILE` to interpolate.

When you run this command, the GDK CLI interpolates all of the variables from the GDK project configuration and builds the `gg-e2e-tests` module to generate the final testing JAR file.

Synopsis

```
$ gdk test-e2e build
```

Arguments

None

Output

The following example shows the output produced when you run this command.

```
$ gdk test-e2e build
[2023-07-20 15:36:48] INFO - Updating feature file: file:///path/to//
HelloWorld/greengrass-build/gg-e2e-tests/src/main/resources/greengrass/features/
component.feature
[2023-07-20 15:36:48] INFO - Creating the E2E testing recipe file:///path/to/
HelloWorld/greengrass-build/recipes/e2e_test_recipe.yaml
[2023-07-20 15:36:48] INFO - Building the E2E testing module
[2023-07-20 15:36:48] INFO - Running the build command 'mvn package'
.........
```

run

Run the testing module with the testing options in the GDK configuration file.

> **Note**
>
> You must build the testing module by running **gdk test-e2e build** before running the end-to-end tests.

Synopsis

```
$ gdk test-e2e run
    [--gtf-options]
```

Arguments

- `-oo, --gtf-options` – (Optional) Specify a list of options for running the end-to-end tests. The argument must be a valid JSON string or a file path to a JSON file containing the GTF options. The options provided in the configuration file are merged with the ones provided in the command arguments. If an option is present in both places, the one in argument takes precendence over the one from the configuration file.

  If the `tags` option is not specified in this command, GDK uses `Sample` for tags. If `ggc-archive` is not specified, GDK downloads the latest version of the Greengrass nucleus archive.

Output

The following example shows the output produced when you run this command.

```
$ gdk test-e2e run
[2023-07-20 16:35:53] INFO - Downloading latest nucleus archive from url https://
d2s8p88vqu9w66.cloudfront.net/releases/greengrass-latest.zip
[2023-07-20 16:35:57] INFO - Running test jar with command java -jar /path/to/
greengrass-build/gg-e2e-tests/target/uat-features-1.0.0.jar –ggc-archive=/path/to/
aws-greengrass-gdk-cli/HelloWorld/greengrass-build/greengrass-nucleus-latest.zip –
tags=Sample

16:35:59.693 [] [] [] [INFO]
 com.aws.greengrass.testing.modules.GreengrassContextModule - Extracting /path/
to/workplace/aws-greengrass-gdk-cli/HelloWorld/greengrass-build/greengrass-
nucleus-latest.zip into /var/folders/7g/ltzcb_3s77nbtmkzfb6brwv40000gr/T/gg-
testing-7718418114158172636/greengrass
16:36:00.534 [gtf-1.1.0-SNAPSHOT] [] [] [INFO]
 com.aws.greengrass.testing.features.LoggerSteps - GTF Version is gtf-1.1.0-SNAPSHOT
.......
```

Greengrass Development Kit CLI configuration file

The AWS IoT Greengrass Development Kit Command-Line Interface (GDK CLI) reads from a configuration file named `gdk-config.json` to build and publish components. This configuration file must exist in the root of the component repository. You can use the GDK CLI init command to initialize component repositories with this configuration file.

Topics

- GDK CLI configuration file format
- GDK CLI configuration file examples

GDK CLI configuration file format

When you define a GDK CLI configuration file for a component, you specify the following information in JSON format.

`gdk_version`

The minimum version of the GDK CLI that is compatible with this component. This value must be one of the GDK CLI versions from releases.

`component`

The configuration for this component.

`componentName`

`author`

The author or publisher of the component.

`version`

The version of the component. Specify one of the following:

- `NEXT_PATCH` – When you choose this option, the GDK CLI sets the version when you publish the component. The GDK CLI queries the AWS IoT Greengrass service to identify the latest published version of the component. Then, it sets the version to the next patch version after that version. If you haven't published the component before, the GDK CLI uses version `1.0.0`.

  If you choose this option, you can't use the Greengrass CLI to locally deploy and test the component to your local development computer that runs the AWS IoT Greengrass Core software. To enable local deployments, you must specify a semantic version instead.

- A semantic version, such as **`1.0.0`**. Semantic versions use a *major.minor.patch* numbering system. For more information, see the semantic version specification.

  If you develop components on a Greengrass core device where you want to deploy and test the component, choose this option. You must build the component with a specific version to create local deployments with the Greengrass CLI.

`build`

The configuration to use to build this component's source into artifacts. This object contains the following information:

`build_system`

The build system to use. Choose from the following options:

- `zip` – Packages the component's folder into a ZIP file to define as the component's only artifact. Choose this option for the following types of components:
  - Components that use interpreted programming languages, such as Python or JavaScript.
  - Components that package files other than code, such as machine learning models or other resources.

  The GDK CLI zips the component's folder into a zip file with the same name as the component folder. For example, if the component folder's name is `HelloWorld`, the GDK CLI creates a zip file named `HelloWorld.zip`.

  > **Note**
  >
  > If you use GDK CLI version 1.0.0 on a Windows device, the component folder and zip file names must contain only lowercase letters.

  When the GDK CLI zips the component's folder into a zip file, it skips the following files:

  - The `gdk-config.json` file
  - The recipe file (`recipe.json` or `recipe.yaml`)
  - Build folders, such as `greengrass-build`
- `maven` – Runs the `mvn clean package` command to build the component's source into artifacts. Choose this option for components that use Maven, such as Java components.

  On Windows devices, this feature is available for GDK CLI v1.1.0 and later.

- `gradle` – Runs the `gradle build` command to build the component's source into artifacts. Choose this option for components that use Gradle. This feature is available for GDK CLI v1.1.0 and later.

  The `gradle` build system supports Kotlin DSL as the build file. This feature is available for GDK CLI v1.2.0 and later.
- `gradlew` – Runs the `gradlew` command to build the component's source into artifacts. Choose this option for components that use the Gradle Wrapper .

  This feature is available for GDK CLI v1.2.0 and later.
- `custom` – Runs a custom command to build the component's source into a recipe and artifacts. Specify the custom command in the `custom_build_command` parameter.

`custom_build_command`

(Optional) The custom build command to run for a custom build system. You must specify this parameter if you specify `custom` for `build_system`.

> **Important**
>
> This command must create a recipe and artifacts in the following folders within the component folder. The GDK CLI creates these folders for you when you run the component build command.
>
> - Recipe folder: `greengrass-build/recipes`
> - Artifacts folder: `greengrass-build/artifacts/componentName/componentVersion`
>
>   Replace *componentName* with the component name, and replace *componentVersion* with the component version or `NEXT_PATCH`.

You can specify a single string or a list of strings, where each string is a word in the command. For example, to run a custom build command for a C++ component, you might specify **`cmake --build build --config Release`** or **`["cmake", "--build", "build", "--config", "Release"]`**.

To view an example of a custom build system, see the aws.greengrass.labs.LocalWebServer community component on GitHub.

`options`

(Optional) Additional configuration options used during the component build process.

This feature is available for GDK CLI v1.2.0 and later.

`excludes`

A list of glob patterns that define which files to exclude from the component directory when building the zip file. Only valid when the `build_system` is `zip`.

> **Note**
>
> In GDK CLI versions 1.4.0 and earlier, any file that matches an entry in the excludes list is excluded from all of the component's subdirectories. To achieve the same behavior in GDK CLI versions 1.5.0 and later, prepend `**/` to the existing entries in the excludes list. For example, `*.txt` will exclude text files from just the directory; `**/*.txt` will exclude text files from all directories and subdirectories.
> In GDK CLI versions 1.5.0 and later, you may see a warning during the component build when `excludes` is defined in the GDK configuration file. To disable this warning, set the environment variable `GDK_EXCLUDES_WARN_IGNORE` to `true`.

The GDK CLI always excludes the following files from the zip file:

- The `gdk-config.json` file
- The recipe file (`recipe.json` or `recipe.yaml`)
- Build folders, such as `greengrass-build`

The following files are excluded by default. However, you can control which of these files are excluded with the `excludes` option.

- Any folder that starts with the prefix "test" (`test*`)
- All hidden files
- The `node_modules` folder

If you specify the `excludes` option, the GDK CLI excludes only those files you set with the `excludes` option. If you don't specify the `excludes` option, the GDK CLI excludes the previously noted default files and folders.

`zip_name`

The zip file name to use when you create a zip artifact during the build process. Only valid when the `build_system` is `zip`. If the `build_system` is empty, the component name is used for the zip file name.

`publish`

The configuration to use to publish this component to the AWS IoT Greengrass service.

If you use GDK CLI v1.1.0 or later, you can specify the `--bucket` argument to specify the S3 bucket where the GDK CLI uploads the component's artifacts. If you don't specify this argument, the GDK CLI uploads to the S3 bucket whose name is *`bucket`*`-`*`region`*`-`*`accountId`*, where *`bucket`* and *`region`* are the values that you specify in `gdk-config.json`, and *`accountId`* is your AWS account ID. The GDK CLI creates the bucket if it doesn't exist.

This object contains the following information:

`bucket`

The S3 bucket name to use to host component artifacts.

`region`

The AWS Region where the GDK CLI publishes this component.

This property is optional if you are using GDK CLI v1.3.0 or later.

`options`

(Optional) Additional configuration options used during component version creation.

This feature is available for GDK CLI v1.2.0 and later.

`file_upload_args`

A JSON structure containing arguments sent to Amazon S3 while uploading files to a bucket, such as metadata and encryption mechanisms. For a list of the allowed arguments, see the `S3Transfer` class in the *Boto3 documentation.*.

`test-e2e`

(Optional) The configuration to use during end-to-end testing of the component. This feature is available for GDK CLI v1.3.0 and later.

`build`

`build_system` – The build system to use. Default option is `maven`. Choose from the following options:

- `maven` – Runs the `mvn package` command to build the testing module. Choose this option for building the testing module that uses Maven.
- `gradle` – Runs the `gradle build` command to build the testing module. Choose this option for the testing module that uses Gradle.

`gtf_version`

(Optional) The version of the Greengrass Testing Framework (GTF) to use as a dependency of the end-to-end testing module when you initialize the GDK project with GTF. This value must be one of the GTF versions from releases. The default is GTF version 1.1.0.

`gtf_options`

(Optional) Additional configuration options used during the end-to-end testing of the component.

The following list includes the options you can use with GTF version 1.1.0.

- `additional-plugins` – (Optional) Additional Cucumber plugins
- `aws-region` – Targets specific regional endpoints for AWS services. Defaults to what the AWS SDK discovers.
- `credentials-path` – Optional AWS profile credentials path. Defaults to credentials discovered on host environment.
- `credentials-path-rotation` – Optional rotation duration for AWS credentials. Defaults to 15 minutes or `PT15M`.
- `csr-path` – The path for the CSR using which the device certificate will be generated.
- `device-mode` – The target device under test. Defaults to local device.
- `env-stage` – Targets the deployment environment of Greengrass. Defaults to production.
- `existing-device-cert-arn` – The arn of an existing certificate that you want to use as a device certificate for Greengrass.
- `feature-path` – File or directory containing additional feature files. Default is no additional feature files are used.
- `gg-cli-version` – Overrides the version of the Greengrass CLI. Defaults to the value found in `ggc.version`.

- `gg-component-bucket` – The name of an existing Amazon S3 bucket that houses Greengrass components.
- `gg-component-overrides` – A list of Greengrass component overrides.
- `gg-persist` – A list of test elements to persist after a test run. Default behavior is to persist nothing. Accepted values are: `aws.resources`, `installed.software`, and `generated.files`.
- `gg-runtime` – A list of values to influence how the test interacts with testing resources. These values supersede the `gg.persist` parameter. If the default is empty, it assumes all testing resources are manged by test case, including the installed Greengrass runtime. Accepted values are: `aws.resources`, `installed.software`, and `generated.files`.
- `ggc-archive` – The path to the archived Greengrass nucleus component.
- `ggc-install-root` – Directory to install the Greengrass nucleus component. Defaults to test.temp.path and test run folder.
- `ggc-log-level` – Set the Greengrass nucleus log level for the test run. Default is "INFO".
- `ggc-tes-rolename` – The IAM role that AWS IoT Greengrass Core will assume to access AWS services. If a role with given name does not exist then one will be created and default access policy.
- `ggc-trusted-plugins` – The comma separate list of the paths (on host) of the trusted plugins that need to added to Greengrass. To provide the path on the DUT itself, prefix the path with 'dut:'
- `ggc-user-name` – The user:group posixUser value for the Greengrass nucleus. Defaults to the current username that is logged in.
- `ggc-version` – Overrides the version of the running Greengrass nucleus component. Defaults to the value found in ggc.archive.
- `log-level` – Log level of the test run. Defaults to "INFO".
- `parallel-config` – Set of batch index and number of batches as a JSON String. Default value of batch index is 0 and number of batches is 1.
- `proxy-url` – Configure all tests to route traffic through this URL.
- `tags` – Only run feature tags. Can be intersected with '&'
- `test-id-prefix` – A common prefix applied to all test specific resources including AWS resource names and tags. Default is a "gg" prefix.
- `test-log-path` – Directory that will contain the results of the entire test run. Defaults to "testResults".

- `test-results-json` – Flag to determine if a resulting Cucumber JSON report is generated written to disk. Defaults to true.
- `test-results-log` – Flag to determine if the console output is generated written to disk. Defaults to false.
- `test-results-xml` – Flag to determine if a resulting JUnit XML report is generated written to disk. Defaults to true.
- `test-temp-path` – Directory to generate local test artifacts. Defaults to a random temp directory prefixed with gg-testing.
- `timeout-multiplier` – Multiplier provided to all test timeouts. Default is 1.0.

GDK CLI configuration file examples

You can reference the following GDK CLI configuration file examples to help you configure Greengrass component environments.

Hello World (Python)

The following GDK CLI configuration file supports a Hello World component that runs a Python script. This configuration file uses the `zip` build system to package the component's Python script into a ZIP file that the GDK CLI uploads as an artifact.

```
{
  "component": {
    "com.example.PythonHelloWorld": {
      "author": "Amazon",
      "version": "NEXT_PATCH",
      "build": {
        "build_system" : "zip",
        "options": {
           "excludes": [".*"]
        }
      },
      "publish": {
        "bucket": "greengrass-component-artifacts",
        "region": "us-west-2",
        "options": {
           "file_upload_args": {
              "Metadata": {
                 "some-key": "some-value"
              }
```

```
            }
          }
        }
      },
  "test-e2e":{
    "build":{
        "build_system": "maven"
    },
    "gtf_version": "1.1.0",
    "gtf_options": {
         "tags": "Sample"
     }
  },
  "gdk_version": "1.6.1"
  }
}
```

Hello World (Java)

The following GDK CLI configuration file supports a Hello World component that runs a Java application. This configuration file uses the `maven` build system to package the component's Java source code into a JAR file that the GDK CLI uploads as an artifact.

```
{
  "component": {
    "com.example.JavaHelloWorld": {
      "author": "Amazon",
      "version": "NEXT_PATCH",
      "build": {
        "build_system" : "maven"
      },
      "publish": {
        "bucket": "greengrass-component-artifacts",
        "region": "us-west-2",
        "options": {
           "file_upload_args": {
              "Metadata": {
                 "some-key": "some-value"
              }
           }
        }
      }
  },
```

```
  "test-e2e":{
    "build":{
        "build_system": "maven"
    },
    "gtf_version": "1.1.0",
    "gtf_options": {
         "tags": "Sample"
     }
  },
  "gdk_version": "1.6.1"
  }
}
```

Community components

Several community components in the Greengrass Software Catalog use the GDK CLI. You can explore the GDK CLI configuration files in these components' repositories.

To view community components' GDK CLI configuration files

1. Run the following command to list the community components that use the GDK CLI.

   ```
   gdk component list --repository
   ```

   The response lists the name of the GitHub repository for each community component that uses the GDK CLI. Each repository exists in the `awslabs` organization.

   ```
   [2022-02-22 17:27:31] INFO - Listing all the available component repositories from
    Greengrass Software Catalog.
   [2022-02-22 17:27:31] INFO - Found '6' component repositories to display.
   1. aws-greengrass-labs-database-influxdb
   2. aws-greengrass-labs-telemetry-influxdbpublisher
   3. aws-greengrass-labs-dashboard-grafana
   4. aws-greengrass-labs-dashboard-influxdb-grafana
   5. aws-greengrass-labs-local-web-server
   6. aws-greengrass-labs-lookoutvision-gstreamer
   ```

2. Open a community component's GitHub repository at the following URL. Replace *community-component-name* with the name of a community component from the previous step.

   ```
   https://github.com/awslabs/community-component-name
   ```

Greengrass Command Line Interface

The Greengrass Command Line Interface (CLI) lets you interact with AWS IoT Greengrass Core on your device to locally develop components and debug issues. For example, you can use the Greengrass CLI to create a local deployment and restart a component on the core device.

Deploy the Greengrass CLI component (`aws.greengrass.Cli`) to install the Greengrass CLI on your core device.

> **Important**
>
> We recommend that you use this component in only development environments, not production environments. This component provides access to information and operations that you typically won't need in a production environment. Follow the principle of least privilege by deploying this component to only core devices where you need it.

Topics

- Install the Greengrass CLI
- Greengrass CLI commands

Install the Greengrass CLI

You can install the Greengrass CLI in one of the following ways:

- Use the `--deploy-dev-tools` argument when you first set up AWS IoT Greengrass Core software on your device. You must also specify `--provision true` to apply this argument.
- Deploy the Greengrass CLI component (`aws.greengrass.Cli`) on your device.

This section describes the steps to deploy the Greengrass CLI component. For information about installing the Greengrass CLI during initial setup, see Tutorial: Getting started with AWS IoT Greengrass V2.

Prerequisites

To deploy the Greengrass CLI component, you must meet the following requirements:

- AWS IoT Greengrass Core software installed and configured on your core device. For more information, see Tutorial: Getting started with AWS IoT Greengrass V2.
- To use the AWS CLI to deploy the Greengrass CLI, you must have installed and configured the AWS CLI. For more information, see Configuring the AWS CLI in the *AWS Command Line Interface User Guide*.
- You must be authorized to use the Greengrass CLI to interact with the AWS IoT Greengrass Core software. Do one of the following to use the Greengrass CLI:
  - Use the system user that runs the AWS IoT Greengrass Core software.
  - Use a user with root or adminstrative permissions. On Linux core devices, you can use `sudo` to gain root permissions.
  - Use a system user that's in a group that you specify in the `AuthorizedPosixGroups` or `AuthorizedWindowsGroups` configuration parameters when you deploy the component. For more information, see Greengrass CLI component configuration.

Deploy the Greengrass CLI component

Complete the following steps to deploy the Greengrass CLI component to your core device:

To deploy the Greengrass CLI component (console)

1. Sign in to the AWS IoT Greengrass console.
2. In the navigation menu, choose **Components**.
3. On the **Components** page, on the **Public components** tab, choose `aws.greengrass.Cli`.
4. On the **aws.greengrass.Cli** page, choose **Deploy**.
5. From **Add to deployment**, choose **Create new deployment**.
6. On the **Specify target** page, under **Deployment targets**, in the **Target name** list, choose the Greengrass group that you want to deploy to, and choose **Next**.
7. On the **Select components** page, verify that the **aws.greengrass.Cli** component is selected, and choose **Next**.
8. On the **Configure components** page, keep the default configuration settings, and choose **Next**.
9. On the **Configure advanced setting** page, keep the default configuration settings, and choose **Next**.
10. On the **Review** page, click **Deploy**

To deploy the Greengrass CLI component (AWS CLI)

1. On your device, create a `deployment.json` file to define the deployment configuration for the Greengrass CLI component. This file should look like the following:

```
{
  "targetArn":"targetArn",
  "components": {
    "aws.greengrass.Cli": {
      "componentVersion": "2.18.4",
      "configurationUpdate": {
        "merge": "{\"AuthorizedPosixGroups\":\"<group1>,<group2>,...,<groupN>\",
\"AuthorizedWindowsGroups\":\"<group1>,<group2>,...,<groupN>\"}"
      }
    }
  }
}
```

   - In the `target` field, replace *`targetArn`* with the Amazon Resource Name (ARN) of the thing or thing group to target for the deployment, in the following format:
     - Thing: `arn:aws:iot:`*`region`*`:`*`account-id`*`:thing/`*`thingName`*
     - Thing group: `arn:aws:iot:`*`region`*`:`*`account-id`*`:thinggroup/`*`thingGroupName`*
   - In the `aws.greengrass.Cli` component object, specify values as follows:

     `version`

     The version of the Greengrass CLI component.

     `configurationUpdate.AuthorizedPosixGroups`

     (Optional) A string that contains a comma-separated list of system groups. You authorize these system groups to use the Greengrass CLI to interact with the AWS IoT Greengrass Core software. You can specify group names or group IDs. For example, `group1,1002,group3` authorizes three system groups (`group1`, `1002`, and `group3`) to use the Greengrass CLI.

     If you don't specify any groups to authorize, you can use the Greengrass CLI as the root user (`sudo`) or as the system user that runs the AWS IoT Greengrass Core software.

`configurationUpdate.AuthorizedWindowsGroups`

(Optional) A string that contains a comma-separated list of system groups. You authorize these system groups to use the Greengrass CLI to interact with the AWS IoT Greengrass Core software. You can specify group names or group IDs. For example, `group1,1002,group3` authorizes three system groups (`group1`, `1002`, and `group3`) to use the Greengrass CLI.

If you don't specify any groups to authorize, you can use the Greengrass CLI as an administrator or as the system user that runs the AWS IoT Greengrass Core software.

2. Run the following command to deploy the Greengrass CLI component on the device:

```
$ aws greengrassv2 create-deployment --cli-input-json file://path/
to/deployment.json
```

During installation, the component adds a symbolic link to `greengrass-cli` in the */greengrass/v2*`/bin` folder on your device, and you run the Greengrass CLI from this path. To run the Greengrass CLI without its absolute path, add your */greengrass/v2*`/bin` folder to your PATH variable. To verify the Greengrass CLI installation, run the following command:

Linux or Unix

```
/greengrass/v2/bin/greengrass-cli help
```

Windows

```
C:\greengrass\v2\bin\greengrass-cli help
```

You should see the following output:

```
Usage: greengrass-cli [-hV] [--ggcRootPath=<ggcRootPath>] [COMMAND]
Greengrass command line interface

      --ggcRootPath=<ggcRootPath>
                  The AWS IoT Greengrass V2 root directory.
  -h, --help      Show this help message and exit.
  -V, --version   Print version information and exit.
Commands:
```

```
  help                Show help information for a command.
  component           Retrieve component information and stop or restart
                        components.
  deployment          Create local deployments and retrieve deployment status.
  logs                Analyze Greengrass logs.
  get-debug-password  Generate a password for use with the HTTP debug view
                        component.
```

If the `greengrass-cli` isn't found, the deployment might have failed to install the Greengrass CLI. For more information, see Troubleshooting AWS IoT Greengrass V2.

Greengrass CLI commands

The Greengrass CLI provides a command line interface to interact locally with your AWS IoT Greengrass core device. Greengrass CLI commands use the following format.

```
$ greengrass-cli <command> <subcommand> [arguments]
```

By default, the `greengrass-cli` executable file in the */greengrass/v2*`/bin/` folder interacts with the version of the AWS IoT Greengrass Core software running in the */greengrass/v2* folder. If you call an executable that is not placed in this location, or if you want to interact with AWS IoT Greengrass Core software in a different location, then you must use one of the following methods to explicitly specify the root path of the AWS IoT Greengrass Core software that you want to interact with:

- Set the `GGC_ROOT_PATH` environment variable to */greengrass/v2*.
- Add the `--ggcRootPath` */greengrass/v2* argument to your command as shown in the following example.

  ```
  greengrass-cli --ggcRootPath /greengrass/v2 <command> <subcommand> [arguments]
  ```

You can use the following arguments with any command:

- Use `--help` for information about a specific Greengrass CLI command.
- Use `--version` for information about the Greengrass CLI version.

This section describes the Greengrass CLI commands and provides examples for these commands. The synopsis for each command shows its arguments and their usage. Optional arguments are shown in square brackets.

Available commands

- component
- deployment
- logs
- get-debug-password

component

Use the `component` command to interact with local components on your core device.

Subcommands

- details
- list
- restart
- stop

details

Retrieve the version, status, and configuration of one component.

Synopsis

```
greengrass-cli component details --name <component-name>
```

Arguments

`--name`, `-n`. The component name.

Output

The following example shows the output produced when you run this command.

```
$ sudo greengrass-cli component details --name MyComponent
```

```
Component Name: MyComponent
Version: 1.0.0
State: RUNNING
Configuration: null
```

list

Retrieve the name, version, status, and configuration of each component installed on the device.

Synopsis

```
greengrass-cli component list
```

Arguments

None

Output

The following example shows the output produced when you run this command.

```
$ sudo greengrass-cli component list

Components currently running in Greengrass:
Component Name: FleetStatusService
Version: 0.0.0
State: RUNNING
Configuration: {"periodicUpdateIntervalSec":86400.0}
Component Name: UpdateSystemPolicyService
Version: 0.0.0
State: RUNNING
Configuration: null
Component Name: aws.greengrass.Nucleus
Version: 2.0.0
State: FINISHED
Configuration: {"awsRegion":"region","runWithDefault":
{"posixUser":"ggc_user:ggc_group"},"telemetry":{}}
Component Name: DeploymentService
Version: 0.0.0
State: RUNNING
Configuration: null
Component Name: TelemetryAgent
```

```
Version: 0.0.0
State: RUNNING
Configuration: null
Component Name: aws.greengrass.Cli
Version: 2.0.0
State: RUNNING
Configuration: {"AuthorizedPosixGroups":"ggc_user"}
```

restart

Restart components.

Synopsis

```
greengrass-cli component restart --names <component-name>,...
```

Arguments

`--names`, `-n`. The component name. At least one component name is required. You can specify additional component names, separating each name with a comma.

Output

None

stop

Stop running components.

Synopsis

```
greengrass-cli component stop --names <component-name>,...
```

Arguments

`--names`, `-n`. The component name. At least one component name is required. You can specify additional component names if needed, separating each name with a comma.

Output

None

deployment

Use the `deployment` command to interact with local components on your core device.

To monitor the progress of a local deployment, use the `status` subcommand. You can't monitor the progress of a local deployment using the console.

Subcommands

- create
- cancel
- list
- status

create

Create or update a local deployment using specified component recipes, artifacts, and runtime arguments.

Synopsis

```
greengrass-cli deployment create
    --recipeDir path/to/component/recipe
    [--artifactDir path/to/artifact/folder ]
    [--update-config {component-configuration}]
    [--groupId <thing-group>]
    [--merge "<component-name>=<component-version>"]...
    [--runWith "<component-name>:posixUser=<user-name>[:<group-name>]"]...
    [--systemLimits "{component-system-resource-limits}]"]...
    [--remove <component-name>,...]
    [--failure-handling-policy <policy name[ROLLBACK, DO_NOTHING]>]
```

Arguments

- `--recipeDir`, `-r`. The full path to the folder that contains the component recipe files.
- `--artifactDir`, `-a`. The full path to the folder that contains the artifact files you want to include in your deployment. The artifacts folder must contain the following directory structure:

  ```
  /path/to/artifact/folder/<component-name>/<component-version>/<artifacts>
  ```

- `--update-config`, `-c`. The configuration arguments for the deployment, provided as a JSON string or a JSON file. The JSON string should be in the following format:

  ```
  { \
    "componentName": { \
      "MERGE": {"config-key": "config-value"}, \
      "RESET": ["path/to/reset/"] \
    } \
  }
  ```

  `MERGE` and `RESET` are case-sensitive and must be in upper case.

- `--groupId`, `-g`. The target thing group for the deployment.
- `--merge`, `-m`. The name and version of the target component that you want to add or update. You must provide the component information in the format *`<component>`*=*`<version>`*. Use a separate argument for each additional component to specify. If needed, use the `--runWith` argument to provide the `posixUser`, `posixGroup`, and `windowsUser` information for running the component.
- `--runWith`. The `posixUser`, `posixGroup`, and `windowsUser` information for running a generic or Lambda component. You must provide this information in the format *`<component>`*`:{posixUser|windowsUser}=`*`<user>`*`[:`*`<=posixGroup>`*`]`. For example, you might specify **`HelloWorld:posixUser=ggc_user:ggc_group`** or **`HelloWorld:windowsUser=ggc_user`**. Use a separate argument for each additional option to specify.

  For more information, see Configure the user that runs components.

- `--systemLimits`. The system resource limits to apply to generic and non-containerized Lambda components' processes on the core device. You can configure the maximum amount of CPU and RAM usage that each component's processes can use. Specify a serialized JSON object or a file path to a JSON file. The JSON object must have the following format.

  ```
  {  \
    "componentName": { \
      "cpus": cpuTimeLimit, \
      "memory": memoryLimitInKb \
    } \
  }
  ```

  You can configure the following system resource limits for each component:

- `cpus` – The maximum amount of CPU time that this component's processes can use on the core device. A core device's total CPU time is equivalent to the device's number of CPU cores. For example, on a core device with 4 CPU cores, you can set this value to 2 to limit this component's processes to 50 percent usage of each CPU core. On a device with 1 CPU core, you can set this value to `0.25` to limit this component's processes to 25 percent usage of the CPU. If you set this value to a number greater than the number of CPU cores, the AWS IoT Greengrass Core software doesn't limit the component's CPU usage.
- `memory` – The maximum amount of RAM (in kilobytes) that this component's processes can use on the core device.

  For more information, see Configure system resource limits for components.

  This feature is available for v2.4.0 and later of the Greengrass nucleus component and Greengrass CLI on Linux core devices. AWS IoT Greengrass doesn't currently support this feature on Windows core devices.

- `--remove`. The name of the target component that you want to remove from a local deployment. If the component defines an uninstall lifecycle step, the AWS IoT Greengrass Core software runs the uninstall script before removing the component. To remove a component that was merged from a cloud deployment, you must provide the group ID of the target thing group in the following format:

  Greengrass nucleus v2.4.0 and later

  ```
  --remove <component-name> --groupId <group-name>
  ```

  Earlier than v2.4.0

  ```
  --remove <component-name> --groupId thinggroup/<group-name>
  ```

- `--failure-handling-policy`. Defines the action taken when a deployment fails. There are two actions that you can specify:
  - `ROLLBACK` – The core device rolls back to its previous configuration when the deployment fails.
  - `DO_NOTHING` – The core device keeps the changes from the failed deployment.

> **Note**
>
> The AWS IoT Greengrass Core software ignores the failure handling policy for bootstrap deployments. A bootstrap deployment updates the Greengrass nucleus version or includes a component with a bootstrap lifecycle step. If a bootstrap deployment fails, the core device rolls back to its previous configuration. The rollback happens even if you set the failure handling policy to `DO_NOTHING`.

This feature is available for v2.11.0 and later of the Greengrass nucleus.

Output

The following example shows the output produced when you run this command.

```
$ sudo greengrass-cli deployment create \
    --merge MyApp1=1.0.0 \
    --merge MyApp2=1.0.0 --runWith MyApp2:posixUser=ggc_user \
    --remove MyApp3 \
    --recipeDir recipes/ \
    --artifactDir artifacts/

Local deployment has been submitted! Deployment Id: 44d89f46-1a29-4044-
ad89-5151213dfcbc
```

cancel

Cancels the specified deployment.

Synopsis

```
greengrass-cli deployment cancel
    -i <deployment-id>
```

Arguments

`-i`. The unique identifier of the deployment to cancel. The deployment ID is returned in the output of the `create` command.

Output

- None

list

Retrieve the status of the last 10 local deployments.

Synopsis

```
greengrass-cli deployment list
```

Arguments

None

Output

The following example shows the output produced when you run this command. Depending on the status of your deployment, the output shows one of the following status values: `IN_PROGRESS`, `SUCCEEDED`, or `FAILED`.

```
$ sudo greengrass-cli deployment list

44d89f46-1a29-4044-ad89-5151213dfcbc: SUCCEEDED
Created on: 6/27/23 11:05 AM
```

status

Retrieve the status of a specific deployment.

Synopsis

```
greengrass-cli deployment status -i <deployment-id>
```

Arguments

`-i`. The ID of the deployment.

Output

The following example shows the output produced when you run this command. Depending on the status of your deployment, the output shows one of the following status values: `IN_PROGRESS`, `SUCCEEDED`, or `FAILED`.

```
$ sudo greengrass-cli deployment status -i 44d89f46-1a29-4044-ad89-5151213dfcbc

44d89f46-1a29-4044-ad89-5151213dfcbc: FAILED
Created on: 6/27/23 11:05 AM
Detailed Status: <Detailed deployment status>
Deployment Error Stack: List of error codes
Deployment Error Types: List of error types
Failure Cause: Cause
```

logs

Use the `logs` command to analyze Greengrass logs on your core device.

Subcommands

- get
- list-keywords
- list-log-files

get

Collect, filter, and visualize Greengrass log files. This command supports only JSON-formatted log files. You can specify the logging format in the nucleus configuration.

Synopsis

```
greengrass-cli logs get
    [--log-dir path/to/a/log/folder]
    [--log-file path/to/a/log/file]
    [--follow true | false ]
    [--filter <filter> ]
    [--time-window <start-time>,<end-time> ]
    [--verbose ]
```

```
    [--no-color ]
    [--before <value> ]
    [--after <value> ]
    [--syslog ]
    [--max-long-queue-size <value> ]
```

Arguments

- `--log-dir`, `-ld`. The path to the directory to check for log files, such as ***/greengrass/v2*/logs**. Do not use with `--syslog`. Use a separate argument for each additional directory to specify. You must use at least one of `--log-dir` or `--log-file`. You can also use both arguments in a single command.
- `--log-file`, `-lf`. The paths to the log directories you want to use. Use a separate argument for each additional directory to specify. You must use at least one of `--log-dir` or `--log-file`. You can also use both arguments in a single command.
- `--follow`, `-fol`. Show log updates as they occur. Greengrass CLI continues to run and reads from the specified logs. If you specify a time window, then Greengrass CLI stops monitoring logs after all of the time windows end.
- `--filter`, `-f`. The keyword, regular expressions, or key-value pair to use as a filter. Provide this value as a string, a regular expression, or as a key-value pair. Use a separate argument for each additional filter to specify.

  When evaluated, multiple filters specified in a single argument are separated by OR operators, and filters specified in additional arguments are combined with AND operators. For example, if your command includes `--filter "installed" --filter "name=alpha,name=beta"`, then Greengrass CLI will filter and display log messages that contain both the keyword `installed` and a `name` key that has the values `alpha` or `beta`.
- `--time-window`, `-t`. The time window for which to show log information. You can use both exact timestamps and relative offsets. You must provide this information in the format *`<begin-time>`*`,`*`<end-time>`*. If you do not specify either the begin time or the end time, then the value for that option defaults to the current system date and time. Use a separate argument for each additional time window to specify.

  Greengrass CLI supports the following formats for timestamps:

  - `yyyy-MM-DD`, for example, `2020-06-30`. The time defaults to 00:00:00 when you use this format.

`yyyyMMDD`, for example, `20200630`. The time defaults to 00:00:00 when you use this format.

`HH:mm:ss`, for example, `15:30:45`. The date defaults to the current system date when you use this format.

`HH:mm:ssSSS`, for example, `15:30:45`. The date defaults the current system date when you use this format.

`YYYY-MM-DD'T'HH:mm:ss'Z'`, for example, `2020-06-30T15:30:45Z`.

`YYYY-MM-DD'T'HH:mm:ss`, for example, `2020-06-30T15:30:45`.

`yyyy-MM-dd'T'HH:mm:ss.SSS`, for example, `2020-06-30T15:30:45.250`.

Relative offsets specify a time period offset from the current system time. Greengrass CLI supports the following format for relative offsets: `+|-[<value>h|hr|hours][value m|min|minutes][value]s|sec|seconds`.

For example, the following argument to specify a time window between 1 hour and 2 hours 15 minutes before the current time is `--time-window -2h15min,-1hr`.

- `--verbose`. Show all fields from the log messages. Do not use with `--syslog`.
- `--no-color`, `-nc`. Remove color coding. The default color coding for log messages uses bold red text. Supports only UNIX-like terminals because it uses ANSI escape sequences.
- `--before`, `-b`. The number of lines to show preceding a matched log entry. Default is 0.
- `--after`, `-a`. The number of lines to show following a matched log entry. Default is 0.
- `--syslog`. Process all log files using the syslog protocol defined by RFC3164. Do not use with `--log-dir` and `--verbose`. The syslog protocol uses the following format: `"<$Priority>$Timestamp $Host $Logger ($Class): $Message"`. If you do not specify a log file, then Greengrass CLI reads log messages from the following locations: `/var/log/messages`, `/var/log/syslog`, or the `/var/log/system.log`.

  AWS IoT Greengrass doesn't currently support this feature on Windows core devices.
- `--max-log-queue-size`, `-m`. The maximum number of log entries to allocate to memory. Use this option to optimize memory usage. Default is 100.

Output

The following example shows the output produced when you run this command.

```
$ sudo greengrass-cli logs get --verbose \
    --log-file /greengrass/v2/logs/greengrass.log \
    --filter deployment,serviceName=DeploymentService \
    --filter level=INFO \
    --time-window 2020-12-08T01:11:17,2020-12-08T01:11:22

2020-12-08T01:11:17.615Z [INFO] (pool-2-thread-14)
 com.aws.greengrass.deployment.DeploymentService: Current deployment finished.
 {DeploymentId=44d89f46-1a29-4044-ad89-5151213dfcbc, serviceName=DeploymentService,
 currentState=RUNNING}
2020-12-08T01:11:17.675Z [INFO] (pool-2-thread-14)
 com.aws.greengrass.deployment.IotJobsHelper: Updating status of persisted
 deployment. {Status=SUCCEEDED, StatusDetails={detailed-deployment-
status=SUCCESSFUL}, ThingName=MyThing, JobId=22d89f46-1a29-4044-ad89-5151213dfcbc
```

list-keywords

Show suggested keywords that you can use to filter log files.

Synopsis

```
greengrass-cli logs list-keywords [arguments]
```

Arguments

None

Output

The following examples show the output produced when you run this command.

```
$ sudo greengrass-cli logs list-keywords

Here is a list of suggested keywords for Greengrass log:
level=$str
thread=$str
loggerName=$str
eventType=$str
```

```
serviceName=$str
error=$str
```

```
$ sudo greengrass-cli logs list-keywords --syslog

Here is a list of suggested keywords for syslog:
priority=$int
host=$str
logger=$str
class=$str
```

list-log-files

Show log files located in a specified directory.

Synopsis

```
greengrass-cli logs list-log-files [arguments]
```

Arguments

`--log-dir`, `-ld`. The path to the directory to check for log files.

Output

The following example shows the output produced when you run this command.

```
$ sudo greengrass-cli logs list-log-files -ld /greengrass/v2/logs/

/greengrass/v2/logs/aws.greengrass.Nucleus.log
/greengrass/v2/logs/main.log
/greengrass/v2/logs/greengrass.log
Total 3 files found.
```

get-debug-password

Use the `get-debug-password` command to print a randomly generated password for the local debug console component (`aws.greengrass.LocalDebugConsole`). The password expires 8 hours after it is generated.

Synopsis

```
greengrass-cli get-debug-password
```

Arguments

None

Output

The following example shows the output produced when you run this command.

```
$ sudo greengrass-cli get-debug-password

Username: debug
Password: bEDp3MOHdj8ou2w5de_sCBI2XAaguy3a8XxREXAMPLE
Password expires at: 2021-04-01T17:01:43.921999931-07:00
The local debug console is configured to use TLS security. The certificate is self-
signed so you will need to bypass your web browser's security warnings to open the
 console.
Before you bypass the security warning, verify that the certificate fingerprint
 matches the following fingerprints.
SHA-256: 15 0B 2C E2 54 8B 22 DE 08 46 54 8A B1 2B 25 DE FB 02 7D 01 4E 4A 56 67 96
 DA A6 CC B1 D2 C4 1B
SHA-1: BC 3E 16 04 D3 80 70 DA E0 47 25 F9 90 FA D6 02 80 3E B5 C1
```

Use AWS IoT Greengrass Testing Framework

Greengrass Testing Framework (GTF) is a collection of building blocks that supports end-to-end automation from the customer perspective. GTF uses Cucumber as the feature driver. AWS IoT Greengrass uses the same building blocks to qualify software changes on various devices. For more information, see Greengrass Testing Framework on Github.

GTF is implemented using Cucumber, a tool used to run automated tests, to encourage a Behavior-Driven Development (BDD) of the components. In Cucumber, the features of this system are outlined in a special type of file called `feature`. Each feature is described in a human-readable format called scenarios which are specifications that can be converted into automated tests. Each scenario is outlined as a series of steps that define the interactions and outcomes of this system under test using a domain-specific language called Gherkin. A Gherkin step is linked to the programming code using a method called step definition which hard wires the specification to the test flow. Step definitions in GTF are implemented with Java.

Topics

- How it works
- Changelog
- Greengrass Testing Framework configuration options
- Tutorial: Run end-to-end tests using Greengrass Testing Framework and Greengrass Development Kit
- Tutorial: Use a confidence test from the confidence test suite

How it works

AWS IoT Greengrass distributes the GTF as a standalone JAR that consists of several Java modules. To use GTF for end-to-end testing of components, you must implement the tests within a Java project. Adding the testing standable JAR as a dependency in your Java project enables you to use the existing functionality of the GTF and extend it by writing your own custom test cases. To run the custom test cases, you can build your Java project and run the target JAR with the configuration options described in Greengrass Testing Framework configuration options.

GTF standalone JAR

Greengrass uses Cloudfront as a Maven repository to host different versions of the GTF standalone JAR. For a full list of GTF versions, see GTF releases.

GTF standalone JAR includes the following modules. It is not limited to only these modules. You can pick and choose each of these dependencies separately in your project or include all of them at once with the testing standalone JAR file.

- `aws-greengrass-testing-resources`: This module provides abstraction for managing the lifecycle of an AWS resource during the course of a test. You can use this to define your custom AWS resources using `ResourceSpec` abstraction so GTF can take care of creation and removal of those resources for you.
- `aws-greengrass-testing-platform`: This module provides platform-level abstraction for the device under test during the test lifecycle. It contains APIs used to interact with the OS independent of the platform and can be used to simulate the commands running in the device shell.
- `aws-greengrass-testing-components`: This module consists of sample components that are used for testing the Greengrass core features such as deployments, IPC, and other features.

- `aws-greengrass-testing-features`: This module consists of reusable common steps and their definitions which are used for testing within in the Greengrass environment.

Topics

- Changelog
- Greengrass Testing Framework configuration options
- Tutorial: Run end-to-end tests using Greengrass Testing Framework and Greengrass Development Kit
- Tutorial: Use a confidence test from the confidence test suite

Changelog

The following table describes the changes in each version of the GTF. For more information, see the GTF Releases page on GitHub.

Version	Changes
1.2.0	New features • Adds network-related steps to configure MQTT and internet network connectivity during tests. • Adds system metric steps to monitor device RAM and CPU use. Bug fixes and improvements • Greengrass CLI local deployment step retries until it succeeds. • Tests gracefully stop Greengrass nucleus instead of killing it. • Adds improvement where GTF polls the AWS IoT Credentials endpoint until credentials are retrievable for the thing and role alias. • Fixes missing artifacts and recipe directories. This version also fixes missing component versions. • Fixes an issue where GTF fails during docker image cleanup if the docker image does not exist. • Adds CURRENT keyword as version of component.

Version	Changes
1.1.0	New features • Adds the ability to install a custom component with configuration. This requires a recipe for the custom component. • Adds the ability to update a local deployment with a custom configura tion. Bug fixes and improvements • Fixes log context GTF version inconsistency issue.
1.0.0	Initial version.

Greengrass Testing Framework configuration options

GTF configuration options

Greengrass Testing Framework (GTF) enables you to configure certain parameters during the launch of the end-to-end testing process to orchestrate the test flow. You can specify these configuration options as CLI arguments for the GTF standalone JAR.

GTF version 1.1.0 and later provides the following configuration options.

- `additional-plugins` – (Optional) Additional Cucumber plugins
- `aws-region` – Targets specific regional endpoints for AWS services. Defaults to what the AWS SDK discovers.
- `credentials-path` – Optional AWS profile credentials path. Defaults to credentials discovered on host environment.
- `credentials-path-rotation` – Optional rotation duration for AWS credentials. Defaults to 15 minutes or `PT15M`.
- `csr-path` – The path for the CSR using which the device certificate will be generated.
- `device-mode` – The target device under test. Defaults to local device.
- `env-stage` – Targets the deployment environment of Greengrass. Defaults to production.
- `existing-device-cert-arn` – The arn of an existing certificate that you want to use as a device certificate for Greengrass.

- `feature-path` – File or directory containing additional feature files. Default is no additional feature files are used.
- `gg-cli-version` – Overrides the version of the Greengrass CLI. Defaults to the value found in `ggc.version`.
- `gg-component-bucket` – The name of an existing Amazon S3 bucket that houses Greengrass components.
- `gg-component-overrides` – A list of Greengrass component overrides.
- `gg-persist` – A list of test elements to persist after a test run. Default behavior is to persist nothing. Accepted values are: `aws.resources`, `installed.software`, and `generated.files`.
- `gg-runtime` – A list of values to influence how the test interacts with testing resources. These values supersede the `gg.persist` parameter. If the default is empty, it assumes all testing resources are manged by test case, including the installed Greengrass runtime. Accepted values are: `aws.resources`, `installed.software`, and `generated.files`.
- `ggc-archive` – The path to the archived Greengrass nucleus component.
- `ggc-install-root` – Directory to install the Greengrass nucleus component. Defaults to test.temp.path and test run folder.
- `ggc-log-level` – Set the Greengrass nucleus log level for the test run. Default is "INFO".
- `ggc-tes-rolename` – The IAM role that AWS IoT Greengrass Core will assume to access AWS services. If a role with given name does not exist then one will be created and default access policy.
- `ggc-trusted-plugins` – The comma separate list of the paths (on host) of the trusted plugins that need to added to Greengrass. To provide the path on the DUT itself, prefix the path with 'dut:'
- `ggc-user-name` – The user:group posixUser value for the Greengrass nucleus. Defaults to the current username that is logged in.
- `ggc-version` – Overrides the version of the running Greengrass nucleus component. Defaults to the value found in ggc.archive.
- `log-level` – Log level of the test run. Defaults to "INFO".
- `parallel-config` – Set of batch index and number of batches as a JSON String. Default value of batch index is 0 and number of batches is 1.
- `proxy-url` – Configure all tests to route traffic through this URL.

- `tags` – Only run feature tags. Can be intersected with '&'
- `test-id-prefix` – A common prefix applied to all test specific resources including AWS resource names and tags. Default is a "gg" prefix.
- `test-log-path` – Directory that will contain the results of the entire test run. Defaults to "testResults".
- `test-results-json` – Flag to determine if a resulting Cucumber JSON report is generated written to disk. Defaults to true.
- `test-results-log` – Flag to determine if the console output is generated written to disk. Defaults to false.
- `test-results-xml` – Flag to determine if a resulting JUnit XML report is generated written to disk. Defaults to true.
- `test-temp-path` – Directory to generate local test artifacts. Defaults to a random temp directory prefixed with gg-testing.
- `timeout-multiplier` – Multiplier provided to all test timeouts. Default is 1.0.

Tutorial: Run end-to-end tests using Greengrass Testing Framework and Greengrass Development Kit

AWS IoT Greengrass Testing Framework (GTF) and Greengrass Development Kit (GDK) offer developers ways to run end-to-end tests. You can complete this tutorial to initialize a GDK project with a component, initialize a GDK project with an end-to-end test module, and build a custom test case. After you build your custom test case, you can then run the test.

In this tutorial, you do the following:

1. Initialize a GDK project with a component.
2. Initialize a GDK project with an end-to-end test module.
3. Build a custom test case.
4. Add a tag to the new test case.
5. Build the test JAR.
6. Run the test.

Topics

- Prerequisites
- Step 1: Initialize a GDK project with a component
- Step 2: Initialize a GDK project with an end-to-end test module
- Step 3: Build a custom test case
- Step 4: Add a tag to the new test case
- Step 5: Build the test JAR
- Step 6: Run the test
- Example: Build a custom test case

Prerequisites

To complete this tutorial, you need the following:

- GDK version 1.3.0 or later
- Java
- Maven
- Git

Step 1: Initialize a GDK project with a component

- Initialize an empty folder with a GDK project. Download the `HelloWorld` component implemented in Python by running the following command.

  ```
  gdk component init -t HelloWorld -l python -n HelloWorld
  ```

  This command creates a new directory named `HelloWorld` in the current directory.

Step 2: Initialize a GDK project with an end-to-end test module

- GDK enables you to download the testing module template consisting of a feature and step implementation. Run the following command to open the `HelloWorld` directory and initialize the existing GDK project using a testing module.

  ```
  cd HelloWorld
  gdk test-e2e init
  ```

This command creates a new directory named `gg-e2e-tests` within the `HelloWorld` directory. This test directory is a Maven project which has a dependency on the Greengrass testing standalone JAR.

Step 3: Build a custom test case

Writing a custom test case broadly consists of two steps: create a feature file with a test scenario and implement step definitions. For an example of building a custom test case, see Example: Build a custom test case. Use the following steps to build your custom test case:

1. Create a feature file with a test scenario

   A feature typically describes a specific functionality of the software that is being tested. In Cucumber, each feature is specified as an individual feature file with a title, a detailed description, and one or more examples of specific cases called scenarios. Each scenario consists of a title, a detailed description, and a series of steps that define the interactions and expected outcomes. Scenarios are written in a structured format using "given," "when," and "then" keywords.
2. Implement step definitions

   A step definition links the Gherkin step in plain language to the programmatic code. When Cucumber identifies a Gherkin step in a scenario, it will look for a matching step definition to run.

Step 4: Add a tag to the new test case

- You can assign tags to the features and scenarios to organize the test process. You can use tags to categorize the subsets of scenarios and also select hooks conditionally to run. Features and scenarios can have multiple tags separated by a space.

  In this example, we are using the `HelloWorld` component.

  In the feature file, add a new tag named `@HelloWorld` beside the `@Sample` tag.

  ```
  @Sample @HelloWorld
  Scenario: As a developer, I can create a component and deploy it on my device
  ....
  ```

Step 5: Build the test JAR

1. Build the component. You must build the component before building the test module.

   ```
   gdk component build
   ```

2. Build the test module using the following command. This command will build the testing JAR in the `greengrass-build` folder.

   ```
   gdk test-e2e build
   ```

Step 6: Run the test

When you run a custom test case, the GTF automates the lifecycle of the test along with managing resources that were created during the test. It first provisions a device under test (DUT) as an AWS IoT thing and installs the Greengrass core software on it. It will then create a new component named `HelloWorld` using the recipe specified in that path. The `HelloWorld` component is then deployed onto the core device through a Greengrass thing deployment. It will then be verified if the deployment is successful. The deployment status will changed to `COMPLETED` within 3 minutes if the deployment is successful.

1. Go to the `gdk-config.json` file in the project directory to target the tests with the `HelloWorld` tag. Update the the `test-e2e` key using the following command.

   ```
   "test-e2e":{
     "gtf_options" : {
          "tags":"HelloWorld"
      }
   }
   ```

2. Before running the tests, you must provide AWS credentials to the host device. GTF uses these credentials to manage the AWS resources during the testing process. Make sure the role you provide has permissions to automate the necessary operations that are included in the test.

   Run the following commands to provide the AWS credentials.

   - Linux or Unix

     ```
     export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
     ```

```
export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

Windows Command Prompt (CMD)

```
set AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
set AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

PowerShell

```
$env:AWS_ACCESS_KEY_ID="AKIAIOSFODNN7EXAMPLE"
$env:AWS_SECRET_ACCESS_KEY="wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"
```

3. Run the test using the following command.

```
gdk test-e2e run
```

This command downloads the latest version of the Greengrass nucleus in the `greengrass-build` folder and runs tests using it. This command also targets only the scenarios with the `HelloWorld` tag and generates a report for those scenarios. You will see the AWS resources that were created during this test are discarded at the end of the test.

Example: Build a custom test case

Example

The downloaded testing module in the GDK project consists of a sample feature and a step implementation file.

In the following example, we create a feature file for testing the thing deployment feature of the Greengrass software. We partially test the functionality of this feature with a scenario that performs deployment of a component through the Greengrass AWS Cloud. This is a series of steps that help us to understand the interactions and expected outcomes of this use case.

1. **Create a feature file**

   Navigate to the `gg-e2e-tests/src/main/resources/greengrass/features` folder in the current directory. You can find the sample `component.feature` that looks like the following example.

In this feature file, you can test the thing deployment feature of the Greengrass software. You can partially test the functionality of this feature with a scenario that performs a deployment of a component through the Greengrass cloud. The scenario is a series of steps that help with understanding the interactions and expected outcomes of this use case.

```
Feature: Testing features of Greengrassv2 component

Background:
    Given my device is registered as a Thing
    And my device is running Greengrass

@Sample
Scenario: As a developer, I can create a component and deploy it on my device
    When I create a Greengrass deployment with components
        HelloWorld | /path/to/recipe/file
    And I deploy the Greengrass deployment configuration
    Then the Greengrass deployment is COMPLETED on the device after 180 seconds
    And I call my custom step
```

GTF contains the step definitions of all of the following steps, except for the step named: `And I call my custom step`.

2. **Implement step definitions**

   GTF standalone JAR contains the step definitions of all of the steps except for one step: `And I call my custom step`. You can implement this step in the testing module.

   Navigate to the source code of the testing file. You can link your custom step using a step definition by using the following command.

```
@And("I call my custom step")
public void customStep() {
    System.out.println("My custom step was called ");
}
```

Tutorial: Use a confidence test from the confidence test suite

AWS IoT Greengrass Testing Framework (GTF) and Greengrass Development Kit (GDK) offer developers ways to run end-to-end tests. You can complete this tutorial to initialize a GDK project

with a component, initialize a GDK project with an end-to-end test module, and use a confidence test from the confidence test suite. After you build your custom test case, you can then run the test.

A confidence test is a generic test provided by Greengrass that validates fundamental component behaviors. These tests can be modified or extended to fit more specific component needs.

For this tutorial we will be using a HelloWorld component. If you are using another component, replace the HelloWorld component with your component.

In this tutorial, you do the following:

1. Initialize a GDK project with a component.
2. Initialize a GDK project with an end-to-end test module.
3. Use a test from the confidence test suite.
4. Add a tag to the new test case.
5. Build the test JAR.
6. Run the test.

Topics

- Prerequisites
- Step 1: Initialize a GDK project with a component
- Step 2: Initialize a GDK project with an end-to-end test module
- Step 3: Use a test from the confidence test suite
- Step 4: Add a tag to the new test case
- Step 5: Build the test JAR
- Step 6: Run the test
- Example: Use a confidence test

Prerequisites

To complete this tutorial, you need the following:

- GDK version 1.6.0 or later

- Java
- Maven
- Git

Step 1: Initialize a GDK project with a component

- Initialize an empty folder with a GDK project. Download the `HelloWorld` component implemented in Python by running the following command.

  ```
  gdk component init -t HelloWorld -l python -n HelloWorld
  ```

  This command creates a new directory named `HelloWorld` in the current directory.

Step 2: Initialize a GDK project with an end-to-end test module

- GDK enables you to download the testing module template consisting of a feature and step implementation. Run the following command to open the `HelloWorld` directory and initialize the existing GDK project using a testing module.

  ```
  cd HelloWorld
  gdk test-e2e init
  ```

  This command creates a new directory named `gg-e2e-tests` within the `HelloWorld` directory. This test directory is a Maven project which has a dependency on the Greengrass testing standalone JAR.

Step 3: Use a test from the confidence test suite

Writing a confidence test case consists of using the provided feature file and, if needed, modifying the scenarios. For an example of using a confidence test, see Example: Build a custom test case. Use the following steps to use a confidence test:

- Use the provided feature file.

  Navigate to `gg-e2e-tests/src/main/resources/greengrass/features` folder in the current directory. Open the sample `confidenceTest.feature` file to use the confidence test.

Step 4: Add a tag to the new test case

- You can assign tags to the features and scenarios to organize the test process. You can use tags to categorize the subsets of scenarios and also select hooks conditionally to run. Features and scenarios can have multiple tags separated by a space.

  In this example, we are using the `HelloWorld` component.

  Each scenario is tagged with `@ConfidenceTest`. Change or add tags if you want to run only a subset of the test suite. Each test scenario is described at the top of each confidence test. The scenario is a series of steps that help with understanding the interactions and expected outcomes of each test case. You can extend these tests by adding your own steps or by modifying the existing ones.

  ```
  @ConfidenceTest
  Scenario: As a Developer, I can deploy GDK_COMPONENT_NAME to my device and see it
   is working as expected
  ....
  ```

Step 5: Build the test JAR

1. Build the component. You must build the component before building the test module.

   ```
   gdk component build
   ```

2. Build the test module using the following command. This command will build the testing JAR in the `greengrass-build` folder.

   ```
   gdk test-e2e build
   ```

Step 6: Run the test

When you run a confidence test, the GTF automates the lifecycle of the test along with managing resources that were created during the test. It first provisions a device under test (DUT) as an AWS IoT thing and installs the Greengrass core software on it. It will then create a new component named `HelloWorld` using the recipe specified in that path. The `HelloWorld` component is then deployed onto the core device through a Greengrass thing deployment. It will then be verified if

the deployment is successful. The deployment status will changed to `COMPLETED` within 3 minutes if the deployment is successful.

1. Go to the `gdk-config.json` file in the project directory to target the tests with the `ConfidenceTest` tag or whichever tag yo8u specified in Step 4. Update the the `test-e2e` key using the following command.

   ```
   "test-e2e":{
     "gtf_options" : {
          "tags":"ConfidenceTest"
      }
   }
   ```

2. Before running the tests, you must provide AWS credentials to the host device. GTF uses these credentials to manage the AWS resources during the testing process. Make sure the role you provide has permissions to automate the necessary operations that are included in the test.

   Run the following commands to provide the AWS credentials.

   - Linux or Unix

     ```
     export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
     export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
     ```

     Windows Command Prompt (CMD)

     ```
     set AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
     set AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
     ```

     PowerShell

     ```
     $env:AWS_ACCESS_KEY_ID="AKIAIOSFODNN7EXAMPLE"
     $env:AWS_SECRET_ACCESS_KEY="wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"
     ```

3. Run the test using the following command.

   ```
   gdk test-e2e run
   ```

   This command downloads the latest version of the Greengrass nucleus in the `greengrass-build` folder and runs tests using it. This command also targets only the scenarios with

the `ConfidenceTest` tag and generates a report for those scenarios. You will see the AWS resources that were created during this test are discarded at the end of the test.

Example: Use a confidence test

Example

The downloaded testing module in the GDK project consists of a provided feature file.

In the following example, we use a feature file for testing the thing deployment feature of the Greengrass software. We partially test the functionality of this feature with a scenario that performs deployment of a component through the Greengrass AWS Cloud. This is a series of steps that help us to understand the interactions and expected outcomes of this use case.

- **Use the provided feature file.**

  Navigate to the `gg-e2e-tests/src/main/resources/greengrass/features` folder in the current directory. You can find the sample `confidenceTest.feature` that looks like the following example.

```
Feature: Confidence Test Suite

Background:
    Given my device is registered as a Thing
    And my device is running Greengrass

@ConfidenceTest
Scenario: As a Developer, I can deploy GDK_COMPONENT_NAME to my device and see it
 is working as expected
    When I create a Greengrass deployment with components
      | GDK_COMPONENT_NAME | GDK_COMPONENT_RECIPE_FILE |
      | aws.greengrass.Cli | LATEST                    |
    And I deploy the Greengrass deployment configuration
    Then the Greengrass deployment is COMPLETED on the device after 180 seconds
    # Update component state accordingly. Possible states: {RUNNING, FINISHED,
 BROKEN, STOPPING}
    And I verify the GDK_COMPONENT_NAME component is RUNNING using the greengrass-
cli
```

  Each test scenario is described at the top of each confidence test. The scenario is a series of steps that help with understanding the interactions and expected outcomes of each test case.

You can extend these tests by adding your own steps or by modifying the existing ones. Each of the scenarios include comments that help you to make these adjustments.

Develop AWS IoT Greengrass components

You can develop and test components on your Greengrass core device. As a result, you can create and iterate your AWS IoT Greengrass software without interacting with the AWS Cloud. When you finish a version of your component, you can upload it to AWS IoT Greengrass in the cloud, so you and your team can deploy the component to other devices in your fleet. For more information about how to deploy components, see Deploy AWS IoT Greengrass components to devices.

Every component is composed of a *recipe* and *artifacts*.

- **Recipes**

  Every component contains a recipe file, which defines its metadata. The recipe also specifies the component's configuration parameters, component dependencies, lifecycle, and platform compatibility. The component lifecycle defines the commands that install, run, and shut down the component. For more information, see AWS IoT Greengrass component recipe reference.

  You can define recipes in JSON or YAML format.

- **Artifacts**

  Components can have any number of artifacts, which are component binaries. Artifacts can include scripts, compiled code, static resources, and any other files that a component consumes. Components can also consume artifacts from component dependencies.

AWS IoT Greengrass provides pre-built components that you can use in your applications and deploy to your devices. For example, you can use the stream manager component to upload data to various AWS services, or you can use the CloudWatch metrics component to publish custom metrics to Amazon CloudWatch. For more information, see AWS-provided components.

AWS IoT Greengrass curates an index of Greengrass components, called the Greengrass Software Catalog. This catalog tracks Greengrass components that are developed by the Greengrass community. From this catalog, you can download, modify, and deploy components to create your Greengrass applications. For more information, see Community components.

The AWS IoT Greengrass Core software runs components as the system user and group, such as `ggc_user` and `ggc_group`, that you configure on the core device. This means that components have the permissions of that system user. If you use a system user without a home directory, then components can't use run commands or code that use a home directory. This means that you can't use the `pip install some-library --user` command to install Python packages for example. If you followed the getting started tutorial to set up your core device, then your system user doesn't have a home directory. For more information about how to configure the user and group that run components, see Configure the user that runs components.

> **Note**
>
> AWS IoT Greengrass uses semantic versions for components. Semantic versions follow a *major.minor.patch* number system. For example, version `1.0.0` represents the first major release for a component. For more information, see the semantic version specification.

Topics

- Component lifecycle
- Component types
- Create AWS IoT Greengrass components
- Test AWS IoT Greengrass components with local deployments
- Publish components to deploy to your core devices
- Interact with AWS services
- Run a Docker container
- AWS IoT Greengrass component recipe reference
- Component environment variable reference

Component lifecycle

The *component lifecycle* defines the stages that the AWS IoT Greengrass Core software uses to install and run components. Each stage defines a script and other information that specifies how the component behaves. For example, when you install a component, the AWS IoT Greengrass Core software runs the `install` lifecycle script for that component. Components on core devices have the following lifecycle states:

- `NEW` – The component's recipe and artifacts are loaded on the core device, but the component isn't installed. After a component enters this state, it runs its install script.
- `INSTALLED` – The component is installed on the core device. The component enters this state after it runs its install script.
- `STARTING` – The component is starting on the core device. The component enters this state when it runs its startup script. If the startup succeeds, the component enters the `RUNNING` state.
- `RUNNING` – The component is running on the core device. The component enters this state when it runs its run script or when it has active background processes from its startup script.
- `FINISHED` – The component ran successfully and completed its run.
- `STOPPING` – The component is stopping. The component enters this state when it runs its shutdown script.
- `ERRORED` – The component encountered an error. When the component enters this state, it runs its recover script. Then, the component restarts to try returning to normal use. If the component enters the `ERRORED` state three times without a successful run, the component becomes `BROKEN`.
- `BROKEN` – The component encountered errors multiple times and can't recover. You must deploy the component again to fix it.
- `UNINSTALLING` – The component is running its uninstall script before permanent removal. The component enters this state when a deployment removes the component from the device. This state does not apply during component version upgrades.
- `UNINSTALLED` – The component completed its uninstall lifecycle step and is removed from the device.

Component types

The *component type* specifies how the AWS IoT Greengrass Core software runs the component. Components can have the following types:

- **Nucleus** (`aws.greengrass.nucleus`)

  The Greengrass nucleus is the component that provides the minimum functionality of the AWS IoT Greengrass Core software. For more information, see Greengrass nucleus.

- **Plugin** (`aws.greengrass.plugin`)

  The Greengrass nucleus runs a plugin component in the same Java Virtual Machine (JVM) as the nucleus. The nucleus restarts when you change the version of a plugin component on a core

device. To install and run plugin components, you must configure the Greengrass nucleus to run as a system service. For more information, see Configure the Greengrass nucleus as a system service.

Several components that are provided by AWS are plugin components, which enables them to interface directly with the Greengrass nucleus. Plugin components use the same log file as the Greengrass nucleus. For more information, see Monitor AWS IoT Greengrass logs.

- **Generic** (`aws.greengrass.generic`)

  The Greengrass nucleus runs a generic component's lifecycle scripts, if the component defines a lifecycle.

  This type is the default type for custom components.

- **Lambda** (`aws.greengrass.lambda`)

  The Greengrass nucleus runs a Lambda function component using the Lambda launcher component.

  When you create a component from a Lambda function, the component has this type. For more information, see Run AWS Lambda functions.

> **Note**
>
> We don't recommend that you specify the component type in a recipe. AWS IoT Greengrass sets the type for you when you create a component.

Create AWS IoT Greengrass components

You can develop custom AWS IoT Greengrass components on a local development computer or a Greengrass core device. AWS IoT Greengrass provides the AWS IoT Greengrass Development Kit Command-Line Interface (GDK CLI) to help you create, build, and publish components from predefined component templates and community components. You can also run built-in shell commands to create, build, and publish components. Choose from the following options to create custom Greengrass components:

- **Use the Greengrass Development Kit CLI**

Use the GDK CLI to develop components on a local development computer. The GDK CLI builds and packages component source code into a recipe and artifacts that you can publish as a private component to the AWS IoT Greengrass service. You can configure the GDK CLI to automatically update the component's version and artifact URIs when you publish the component, so you don't need to update the recipe each time. To develop a component using the GDK CLI, you can start from a template or a community component from the Greengrass Software Catalog. For more information, see AWS IoT Greengrass Development Kit Command-Line Interface.

- **Run built-in shell commands**

  You can run built-in shell commands to develop components on a local development computer or on a Greengrass core device. You use shell commands to copy or build component source code into artifacts. Each time you create a new version of a component, you must create or update the recipe with the new component version. When you publish the component to the AWS IoT Greengrass service, you must update the URI to each component artifact in the recipe.

Topics

- Create a component (GDK CLI)
- Create a component (shell commands)

Create a component (GDK CLI)

Follow instructions in this section to create and build a component using the GDK CLI.

To develop a Greengrass component (GDK CLI)

1. If you haven't already, install the GDK CLI on your development computer. For more information, see Install or update the AWS IoT Greengrass Development Kit Command-Line Interface.
2. Change to the folder where you want to create component folders.

   Linux or Unix

   ```
   mkdir ~/greengrassv2
   cd ~/greengrassv2
   ```

Windows Command Prompt (CMD)

```
mkdir %USERPROFILE%\greengrassv2
cd %USERPROFILE%\greengrassv2
```

PowerShell

```
mkdir ~/greengrassv2
cd ~/greengrassv2
```

3. Choose a component template or community component to download. The GDK CLI downloads the template or community component, so you can start from a functional example. Use the component list command to retrieve the list of available templates or community components.

   - To list component templates, run the following command. Each line in the response includes a template's name and programming language.

     ```
     gdk component list --template
     ```

   - To list community components, run the following command.

     ```
     gdk component list --repository
     ```

4. Create and change to a component folder where the GDK CLI downloads the template or community component. Replace *HelloWorld* with the name of the component, or another name that helps you identify this component folder.

   Linux or Unix

   ```
   mkdir HelloWorld
   cd HelloWorld
   ```

   Windows Command Prompt (CMD)

   ```
   mkdir HelloWorld
   cd HelloWorld
   ```

PowerShell

```
mkdir HelloWorld
cd HelloWorld
```

5. Download the template or community component to the current folder. Use the component init command.

   - To create a component folder from a template, run the following command. Replace *HelloWorld* with the name of the template, and replace *python* with the name of the programming language.

     ```
     gdk component init --template HelloWorld --language python
     ```

   - To create a component folder from a community component, run the following command. Replace *ComponentName* with the name of the community component.

     ```
     gdk component init --repository ComponentName
     ```

   Note

   If you use GDK CLI v1.0.0, you must run this command in an empty folder. The GDK CLI downloads the template or community component to the current folder.
   If you use GDK CLI v1.1.0 or later, you can specify the `--name` argument to specify the folder where the GDK CLI downloads the template or community component. If you use this argument, specify a folder that doesn't exist. The GDK CLI creates the folder for you. If you don't specify this argument, the GDK CLI uses the current folder, which must be empty.

6. The GDK CLI reads from the GDK CLI configuration file, named `gdk-config.json`, to build and publish components. This configuration file exists in the root of the component folder. The previous step creates this file for you. In this step, you update `gdk-config.json` with information about your component. Do the following:

   a. Open `gdk-config.json` in a text editor.

   b. (Optional) Change the name of the component. The component name is the key in the `component` object.

   c. Change the author of the component.

d. (Optional) Change the version of the component. Specify one of the following:

- `NEXT_PATCH` – When you choose this option, the GDK CLI sets the version when you publish the component. The GDK CLI queries the AWS IoT Greengrass service to identify the latest published version of the component. Then, it sets the version to the next patch version after that version. If you haven't published the component before, the GDK CLI uses version `1.0.0`.

  If you choose this option, you can't use the Greengrass CLI to locally deploy and test the component to your local development computer that runs the AWS IoT Greengrass Core software. To enable local deployments, you must specify a semantic version instead.

- A semantic version, such as **`1.0.0`**. Semantic versions use a *major.minor.patch* numbering system. For more information, see the semantic version specification.

  If you develop components on a Greengrass core device where you want to deploy and test the component, choose this option. You must build the component with a specific version to create local deployments with the Greengrass CLI.

e. (Optional) Change the build configuration for the component. The build configuration defines how the GDK CLI builds the component's source into artifacts. Choose from the following options for `build_system`:

- `zip` – Packages the component's folder into a ZIP file to define as the component's only artifact. Choose this option for the following types of components:
  - Components that use interpreted programming languages, such as Python or JavaScript.
  - Components that package files other than code, such as machine learning models or other resources.

  The GDK CLI zips the component's folder into a zip file with the same name as the component folder. For example, if the component folder's name is `HelloWorld`, the GDK CLI creates a zip file named `HelloWorld.zip`.

  > **Note**
  >
  > If you use GDK CLI version 1.0.0 on a Windows device, the component folder and zip file names must contain only lowercase letters.

When the GDK CLI zips the component's folder into a zip file, it skips the following files:

- The `gdk-config.json` file
- The recipe file (`recipe.json` or `recipe.yaml`)
- Build folders, such as `greengrass-build`

- `maven` – Runs the `mvn clean package` command to build the component's source into artifacts. Choose this option for components that use Maven, such as Java components.

  On Windows devices, this feature is available for GDK CLI v1.1.0 and later.

- `gradle` – Runs the `gradle build` command to build the component's source into artifacts. Choose this option for components that use Gradle. This feature is available for GDK CLI v1.1.0 and later.

  The `gradle` build system supports Kotlin DSL as the build file. This feature is available for GDK CLI v1.2.0 and later.

- `gradlew` – Runs the `gradlew` command to build the component's source into artifacts. Choose this option for components that use the Gradle Wrapper .

  This feature is available for GDK CLI v1.2.0 and later.

- `custom` – Runs a custom command to build the component's source into a recipe and artifacts. Specify the custom command in the `custom_build_command` parameter.

f. If you specify `custom` for `build_system`, add the `custom_build_command` to the `build` object. In `custom_build_command`, specify a single string or list of strings, where each string is a word in the command. For example, to run a custom build command for a C++ component, you might specify **`["cmake", "--build", "build", "--config", "Release"]`**.

g. If you use GDK CLI v1.1.0 or later, you can specify the `--bucket` argument to specify the S3 bucket where the GDK CLI uploads the component's artifacts. If you don't specify this argument, the GDK CLI uploads to the S3 bucket whose name is *`bucket`*`-`*`region`*`-`*`accountId`*, where *`bucket`* and *`region`* are the values that you specify in `gdk-config.json`, and *`accountId`* is your AWS account ID. The GDK CLI creates the bucket if it doesn't exist.

   Change the publish configuration for the component. Do the following:

i. Specify the name of the S3 bucket to use to host component artifacts.

ii. Specify the AWS Region where the GDK CLI publishes the component.

When you're done with this step, the `gdk-config.json` file might look similar to the following example.

```
{
  "component": {
    "com.example.PythonHelloWorld": {
      "author": "Amazon",
      "version": "NEXT_PATCH",
      "build": {
        "build_system" : "zip"
      },
      "publish": {
        "bucket": "greengrass-component-artifacts",
        "region": "us-west-2"
      }
    }
  },
  "gdk_version": "1.0.0"
}
```

7. Update the component recipe file, named `recipe.yaml` or `recipe.json`. Do the following:

   a. If you downloaded a template or community component that uses the `zip` build system, check that the zip artifact name matches the name of the component folder. The GDK CLI zips the component folder into a zip file with the same name as the component folder. The recipe contains the zip artifact name in the list of component artifacts and in lifecycle scripts that use files in the zip artifact. Update the `Artifacts` and `Lifecycle` definitions such that the zip file name matches the name of the component folder. The following partial recipe examples highlight the zip file name in the `Artifacts` and `Lifecycle` definitions.

      JSON

      ```
      {
        ...
        "Manifests": [
          {
      ```

```
      "Platform": {
        "os": "all"
      },
      "Artifacts": [
        {
          "URI": "s3://{COMPONENT_NAME}/{COMPONENT_VERSION}/HelloWorld.zip",
          "Unarchive": "ZIP"
        }
      ],
      "Lifecycle": {
        "Run": "python3 -u {artifacts:decompressedPath}/HelloWorld/main.py
 {configuration:/Message}"
      }
    }
  ]
}
```

YAML

```
---
...
Manifests:
  - Platform:
      os: all
    Artifacts:
      - URI: "s3://BUCKET_NAME/COMPONENT_NAME/
COMPONENT_VERSION/HelloWorld.zip"
        Unarchive: ZIP
    Lifecycle:
      Run: "python3 -u {artifacts:decompressedPath}/HelloWorld/main.py
 {configuration:/Message}"
```

b. (Optional) Update the component description, default configuration, artifacts, lifecycle scripts, and platform support. For more information, see AWS IoT Greengrass component recipe reference.

When you're done with this step, the recipe file might look similar to the following examples.

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
```

```
  "ComponentName": "{COMPONENT_NAME}",
  "ComponentVersion": "{COMPONENT_VERSION}",
  "ComponentDescription": "This is a simple Hello World component written in
 Python.",
  "ComponentPublisher": "{COMPONENT_AUTHOR}",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "Message": "World"
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "all"
      },
      "Artifacts": [
        {
          "URI": "s3://{COMPONENT_NAME}/{COMPONENT_VERSION}/HelloWorld.zip",
          "Unarchive": "ZIP"
        }
      ],
      "Lifecycle": {
        "Run": "python3 -u {artifacts:decompressedPath}/HelloWorld/main.py
 {configuration:/Message}"
      }
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: "2020-01-25"
ComponentName: "{COMPONENT_NAME}"
ComponentVersion: "{COMPONENT_VERSION}"
ComponentDescription: "This is a simple Hello World component written in
 Python."
ComponentPublisher: "{COMPONENT_AUTHOR}"
ComponentConfiguration:
  DefaultConfiguration:
    Message: "World"
Manifests:
  - Platform:
```

```
      os: all
    Artifacts:
      - URI: "s3://BUCKET_NAME/COMPONENT_NAME/COMPONENT_VERSION/HelloWorld.zip"
        Unarchive: ZIP
    Lifecycle:
      Run: "python3 -u {artifacts:decompressedPath}/HelloWorld/main.py
 {configuration:/Message}"
```

8. Develop and build the Greengrass component. The component build command produces a recipe and artifacts in the `greengrass-build` folder in the component folder. Run the following command.

```
gdk component build
```

When you're ready to test your component, use the GDK CLI to publish it to the AWS IoT Greengrass service. Then, you can deploy the component to Greengrass core devices. For more information, see Publish components to deploy to your core devices.

Create a component (shell commands)

Follow instructions in this section to create recipe and artifact folders that contain source code and artifacts for multiple components.

To develop a Greengrass component (shell commands)

1. Create a folder for your components with subfolders for recipes and artifacts. Run the following commands on your Greengrass core device to create these folders and change to the component folder. Replace *~/greengrassv2* or *%USERPROFILE%\greengrassv2* with the path to the folder to use for local development.

   Linux or Unix

   ```
   mkdir -p ~/greengrassv2/{recipes,artifacts}
   cd ~/greengrassv2
   ```

   Windows Command Prompt (CMD)

   ```
   mkdir %USERPROFILE%\greengrassv2\\recipes, %USERPROFILE%\greengrassv2\\artifacts
   cd %USERPROFILE%\greengrassv2
   ```

PowerShell

```
mkdir ~/greengrassv2/recipes, ~/greengrassv2/artifacts
cd ~/greengrassv2
```

2. Use a text editor to create a recipe file that defines your component's metadata, parameters, dependencies, lifecycle, and platform capability. Include the component version in the recipe file name so that you can identify which recipe reflects which component version. You can choose YAML or JSON format for your recipe.

   For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

   JSON

   ```
   nano recipes/com.example.HelloWorld-1.0.0.json
   ```

   YAML

   ```
   nano recipes/com.example.HelloWorld-1.0.0.yaml
   ```

   > **Note**
   >
   > AWS IoT Greengrass uses semantic versions for components. Semantic versions follow a *major.minor.patch* number system. For example, version `1.0.0` represents the first major release for a component. For more information, see the semantic version specification.

3. Define the recipe for your component. For more information, see AWS IoT Greengrass component recipe reference.

   Your recipe might look similar to the following Hello World example recipe.

   JSON

   ```
   {
     "RecipeFormatVersion": "2020-01-25",
     "ComponentName": "com.example.HelloWorld",
   ```

```
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "My first AWS IoT Greengrass component.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "Message": "world"
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "run": "python3 -u {artifacts:path}/hello_world.py {configuration:/
Message}"
      }
    },
    {
      "Platform": {
        "os": "windows"
      },
      "Lifecycle": {
        "run": "py -3 -u {artifacts:path}/hello_world.py {configuration:/
Message}"
      }
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.HelloWorld
ComponentVersion: '1.0.0'
ComponentDescription: My first AWS IoT Greengrass component.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    Message: world
Manifests:
  - Platform:
```

```
      os: linux
    Lifecycle:
      run: |
        python3 -u {artifacts:path}/hello_world.py "{configuration:/Message}"
  - Platform:
      os: windows
    Lifecycle:
      run: |
        py -3 -u {artifacts:path}/hello_world.py "{configuration:/Message}"
```

This recipe runs a Hello World Python script, which might look similar to the following example script.

```
import sys

message = "Hello, %s!" % sys.argv[1]

# Print the message to stdout, which Greengrass saves in a log file.
print(message)
```

4. Create a folder for the component version to develop. We recommend that you use a separate folder for each component version's artifacts so that you can identify which artifacts are for each component version. Run the following command.

   Linux or Unix

   ```
   mkdir -p artifacts/com.example.HelloWorld/1.0.0
   ```

   Windows Command Prompt (CMD)

   ```
   mkdir artifacts/com.example.HelloWorld/1.0.0
   ```

   PowerShell

   ```
   mkdir artifacts/com.example.HelloWorld/1.0.0
   ```

> **Important**
>
> You must use the following format for the artifact folder path. Include the component name and version that you specify in the recipe.
>
> ```
> artifacts/componentName/componentVersion/
> ```

5. Create the artifacts for your component in the folder that you created in the previous step. Artifacts can include software, images, and any other binaries that your component uses.

   When your component is ready, test your component.

Test AWS IoT Greengrass components with local deployments

If you develop a Greengrass component on a core device, you can create a local deployment to install and test it. Follow the steps in this section to create a local deployment.

If you develop the component on a different computer, such as a local development computer, you can't create a local deployment. Instead, publish the component to the AWS IoT Greengrass service so that you can deploy it to Greengrass core devices to test it. For more information, see Publish components to deploy to your core devices and Deploy AWS IoT Greengrass components to devices.

To test a component on an Greengrass core device

1. The core device logs events such as component updates. You can view this log file to discover and troubleshoot errors with your component, such as an invalid recipe. This log file also displays messages that your component prints to standard out (stdout). We recommend that you open an additional terminal session on your core device to observe new log messages in real time. Open a new terminal session, such as through SSH, and run the following command to view the logs. Replace */greengrass/v2* with the path to the AWS IoT Greengrass root folder.

   Linux or Unix

   ```
   sudo tail -f /greengrass/v2/logs/greengrass.log
   ```

PowerShell

```
gc C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
```

You can also view the log file for your component.

Linux or Unix

```
sudo tail -f /greengrass/v2/logs/com.example.HelloWorld.log
```

PowerShell

```
gc C:\greengrass\v2\logs\com.example.HelloWorld.log -Tail 10 -Wait
```

2. In your original terminal session, run the following command to update the core device with your component. Replace */greengrass/v2* with the path to the AWS IoT Greengrass root folder, and replace *~/greengrassv2* with the path to your local development folder.

   Linux or Unix

   ```
   sudo /greengrass/v2/bin/greengrass-cli deployment create \
     --recipeDir ~/greengrassv2/recipes \
     --artifactDir ~/greengrassv2/artifacts \
     --merge "com.example.HelloWorld=1.0.0"
   ```

   Windows Command Prompt (CMD)

   ```
   C:\greengrass\v2\bin\greengrass-cli deployment create ^
     --recipeDir %USERPROFILE%\greengrassv2\recipes ^
     --artifactDir %USERPROFILE%\greengrassv2\artifacts ^
     --merge "com.example.HelloWorld=1.0.0"
   ```

   PowerShell

   ```
   C:\greengrass\v2\bin\greengrass-cli deployment create `
     --recipeDir ~/greengrassv2/recipes `
     --artifactDir ~/greengrassv2/artifacts `
     --merge "com.example.HelloWorld=1.0.0"
   ```

> **Note**
>
> You can also use the `greengrass-cli deployment create` command to set the value of your component's configuration parameters. For more information, see create.

3. Use the `greengrass-cli deployment status` command to monitor the progress of your component's deployment.

   Unix or Linux

   ```
   sudo /greengrass/v2/bin/greengrass-cli deployment status \
     -i deployment-id
   ```

   Windows Command Prompt (CMD)

   ```
   C:\greengrass\v2\bin\greengrass-cli deployment status ^
     -i deployment-id
   ```

   PowerShell

   ```
   C:\greengrass\v2\bin\greengrass-cli deployment status `
     -i deployment-id
   ```

4. Test your component as it runs on the Greengrass core device. When you finish this version of your component, you can upload it to the AWS IoT Greengrass service. Then, you can deploy the component to other core devices. For more information, see Publish components to deploy to your core devices.

Publish components to deploy to your core devices

After you build or complete a version of a component, you can publish it to the AWS IoT Greengrass service. Then, you can deploy it to Greengrass core devices.

If you use the Greengrass Development Kit CLI (GDK CLI) to develop and build a component, you can use the GDK CLI to publish the component to the AWS Cloud. Otherwise, use built-in shell commands and the AWS CLI to publish the component.

You can also use AWS CloudFormation to create components and other AWS resources from templates. For more information, see What is AWS CloudFormation? and AWS::GreengrassV2::ComponentVersion in the *AWS CloudFormation User Guide*.

Topics

- Publish a component (GDK CLI)
- Publish a component (shell commands)

Publish a component (GDK CLI)

Follow instructions in this section to publish a component using the GDK CLI. The GDK CLI uploads build artifacts to an S3 bucket, updates the artifact URIs in the recipe, and creates the component from the recipe. You specify the S3 bucket and Region to use in the GDK CLI configuration file.

If you use GDK CLI v1.1.0 or later, you can specify the `--bucket` argument to specify the S3 bucket where the GDK CLI uploads the component's artifacts. If you don't specify this argument, the GDK CLI uploads to the S3 bucket whose name is *`bucket`*`-`*`region`*`-`*`accountId`*, where *`bucket`* and *`region`* are the values that you specify in `gdk-config.json`, and *`accountId`* is your AWS account ID. The GDK CLI creates the bucket if it doesn't exist.

> **Important**
>
> Core device roles don't allow access to S3 buckets by default. If this is your first time using this S3 bucket, you must add permissions to the role to allow core devices to retrieve component artifacts from this S3 bucket. For more information, see Allow access to S3 buckets for component artifacts.

To publish a Greengrass component (GDK CLI)

1. Open the component folder in a command prompt or terminal.
2. If you haven't already, build the Greengrass component. The component build command produces a recipe and artifacts in the `greengrass-build` folder in the component folder. Run the following command.

   ```
   gdk component build
   ```

3. Publish the component to the AWS Cloud. The component publish command uploads the component's artifacts to Amazon S3 and updates the component's recipe with each artifact's URI. Then, it creates the component in the AWS IoT Greengrass service.

   > **Note**
   >
   > AWS IoT Greengrass computes the digest of each artifact when you create the component. This means that you can't modify the artifact files in your S3 bucket after you create a component. If you do, deployments that include this component will fail, because the file digest doesn't match. If you modify an artifact file, you must create a new version of the component.

   If you specify `NEXT_PATCH` for the component version in the GDK CLI configuration file, the GDK CLI uses the next patch version that doesn't already exist in the AWS IoT Greengrass service.

   Run the following command.

   ```
   gdk component publish
   ```

   The output tells you the version of the component that the GDK CLI created.

   After you publish the component, you can deploy the component to core devices. For more information, see Deploy AWS IoT Greengrass components to devices.

Publish a component (shell commands)

Use the following procedure to publish a component using shell commands and the AWS Command Line Interface (AWS CLI). When you publish a component, you do the following:

1. Publish component artifacts to an S3 bucket.
2. Add each artifact's Amazon S3 URI to the component recipe.
3. Create a component version in AWS IoT Greengrass from the component recipe.

> **Note**
>
> Each component version that you upload must be unique. Make sure that you upload the correct component version, because you can't edit it after you upload it.

You can follow these steps to publish a component from your development computer or your Greengrass core device.

To publish a component (shell commands)

1. If the component uses a version that exists in the AWS IoT Greengrass service, then you must change the version of the component. Open the recipe in a text editor, increment the version, and save the file. Choose a new version that reflects the changes that you made to the component.

   > **Note**
   >
   > AWS IoT Greengrass uses semantic versions for components. Semantic versions follow a *major.minor.patch* number system. For example, version `1.0.0` represents the first major release for a component. For more information, see the semantic version specification.

2. If your component has artifacts, do the following:

   a. Publish the component's artifacts to an S3 bucket in your AWS account.

      > **Tip**
      >
      > We recommend that you include the component name and version in the path to the artifact in the S3 bucket. This naming scheme can help you maintain the artifacts that previous versions of the component use, so you can continue to support previous component versions.

      Run the following command to publish an artifact file to an S3 bucket. Replace amzn-s3-demo-bucket with the name of the bucket, and replace *`artifacts/com.example.HelloWorld/1.0.0/artifact.py`* with the path to the artifact file.

```
aws s3 cp artifacts/com.example.HelloWorld/1.0.0/artifact.py s3://amzn-s3-demo-bucket/artifacts/com.example.HelloWorld/1.0.0/artifact.py
```

Tip

To speed up component creation for large artifacts, consider uploading your artifacts to S3 as a single-part upload with SHA-256 checksums enabled. Use the `--checksum-algorithm SHA256` option with the AWS CLI, and set `s3.multipart_threshold` higher than your file size to prevent multipart upload. For example, to upload a 2 GB artifact:

```
aws configure set s3.multipart_threshold 5GB
aws s3 cp artifacts/com.example.HelloWorld/1.0.0/artifact.bin s3://amzn-s3-demo-bucket/artifacts/com.example.HelloWorld/1.0.0/artifact.bin --checksum-algorithm SHA256
```

If your S3 bucket uses a restrictive bucket policy, ensure that the policy allows the `s3:GetObjectAttributes` action so that AWS IoT Greengrass can retrieve the pre-computed checksum.

Important

Core device roles don't allow access to S3 buckets by default. If this is your first time using this S3 bucket, you must add permissions to the role to allow core devices to retrieve component artifacts from this S3 bucket. For more information, see Allow access to S3 buckets for component artifacts.

b. Add a list named `Artifacts` to the component recipe if it isn't present. The `Artifacts` list appears in each manifest, which defines the component's requirements on each platform that it supports (or the component's default requirements for all platforms).

c. Add each artifact to the list of artifacts, or update the URI of existing artifacts. The Amazon S3 URI is composed of the bucket name and the path to the artifact object in the bucket. Your artifacts' Amazon S3 URIs should look similar to the following example.

```
s3://amzn-s3-demo-bucket/artifacts/com.example.HelloWorld/1.0.0/artifact.py
```

After you complete these steps, your recipe should have an `Artifacts` list that looks like the following.

JSON

```
{
  ...
  "Manifests": [
    {
      "Lifecycle": {
        ...
      },
      "Artifacts": [
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
MyGreengrassComponent/1.0.0/artifact.py",
          "Unarchive": "NONE"
        }
      ]
    }
  ]
}
```

> **Note**
>
> You can add the `"Unarchive": "ZIP"` option for a ZIP artifact to configure the AWS IoT Greengrass Core software to unzip the artifact when the component deploys.

YAML

```
...
Manifests:
  - Lifecycle:
      ...
    Artifacts:
      - URI: s3://amzn-s3-demo-bucket/artifacts/MyGreengrassComponent/1.0.0/
artifact.py
```

```
        Unarchive: NONE
```

> **Note**
>
> You can use the `Unarchive: ZIP` option to configure the AWS IoT Greengrass Core software to unzip a ZIP artifact when the component deploys. For more information about how to use ZIP artifacts in a component, see the artifacts:decompressedPath recipe variable.

For more information about recipes, see AWS IoT Greengrass component recipe reference.

3. Use the AWS IoT Greengrass console to create a component from the recipe file.

   Run the following command to create the component from a recipe file. This command creates the component and publishes it as a private AWS IoT Greengrass component in your AWS account. Replace *path/to/recipeFile* with the path to the recipe file.

   ```
   aws greengrassv2 create-component-version --inline-recipe fileb://path/to/recipeFile
   ```

   Copy the `arn` from the response to check the state of the component in the next step.

   > **Note**
   >
   > AWS IoT Greengrass computes the digest of each artifact when you create the component. This means that you can't modify the artifact files in your S3 bucket after you create a component. If you do, deployments that include this component will fail, because the file digest doesn't match. If you modify an artifact file, you must create a new version of the component.

4. Each component in the AWS IoT Greengrass service has a state. Run the following command to confirm the state of the component version that you publish in this procedure. Replace *com.example.HelloWorld* and *1.0.0* with the component version to query. Replace the `arn` with the ARN from the previous step.

   ```
   aws greengrassv2 describe-component --arn "arn:aws:greengrass:region:account-id:components:com.example.HelloWorld:versions:1.0.0"
   ```

The operation returns a response that contains the component's metadata. The metadata contains a `status` object that contains the component state and any errors, if applicable.

When the component state is `DEPLOYABLE`, you can deploy the component to devices. For more information, see Deploy AWS IoT Greengrass components to devices.

Interact with AWS services

Greengrass core devices use X.509 certificates to connect to AWS IoT Core using TLS mutual authentication protocols. These certificates let devices interact with AWS IoT without AWS credentials, which typically comprise an access key ID and a secret access key. Other AWS services require AWS credentials instead of X.509 certificates to call API operations at service endpoints. AWS IoT Core has a credentials provider that enables devices to use their X.509 certificate to authenticate AWS requests. The AWS IoT credentials provider authenticates devices using an X.509 certificate and issues AWS credentials in the form a temporary, limited-privilege security token. Devices can use this token to sign and authenticate any AWS request. This eliminates the need to store AWS credentials on Greengrass core devices. For more information, see Authorizing direct calls to AWS services in the *AWS IoT Core Developer Guide*.

To fetch credentials from AWS IoT, Greengrass, core devices use an AWS IoT role alias that points to an IAM role. This IAM role is called the *token exchange role*. You create the role alias and token exchange role when you install the AWS IoT Greengrass Core software. To specify the role alias that a core device uses, configure the `iotRoleAlias` parameter of the Greengrass nucleus.

The AWS IoT credentials provider assumes the token exchange role on your behalf to provide AWS credentials to core devices. You can attach appropriate IAM policies to this role to allow your core devices access to your AWS resources, such as components artifacts in S3 buckets. For more information about how to configure the token exchange role, see Authorize core devices to interact with AWS services.

Greengrass core devices store AWS credentials in memory, and the credentials expire after an hour by default. If the AWS IoT Greengrass Core software restarts, it must fetch credentials again. You can use the UpdateRoleAlias operation to configure the duration that credentials are valid.

AWS IoT Greengrass provides a public component, the token exchange service component, that you can define as a dependency in your custom component to interact with AWS services. The token exchange service provides your component with an environment variable, `AWS_CONTAINER_CREDENTIALS_FULL_URI`, that defines the URI to a local server that provides

AWS credentials. When you create an AWS SDK client, the client checks for this environment variable and connects to the local server to retrieve AWS credentials and uses them to sign API requests. This lets you use AWS SDKs and other tools to call AWS services in your components. For more information, see Token exchange service.

Important

Support to acquire AWS credentials in this way was added to the AWS SDKs on July 13th, 2016. Your component must use an AWS SDK version that was created on or after that date. For more information, see Using a supported AWS SDK in the *Amazon Elastic Container Service Developer Guide*.

To acquire AWS credentials in your custom component, define `aws.greengrass.TokenExchangeService` as a dependency in the component recipe. The following example recipe defines a component that installs boto3 and runs a Python script that uses AWS credentials from the token exchange service to list Amazon S3 buckets.

Note

To run this example component, your device must have the `s3:ListAllMyBuckets` permission. For more information, see Authorize core devices to interact with AWS services.

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.ListS3Buckets",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that uses the token exchange service to list
 S3 buckets.",
  "ComponentPublisher": "Amazon",
  "ComponentDependencies": {
    "aws.greengrass.TokenExchangeService": {
      "VersionRequirement": "^2.0.0",
      "DependencyType": "HARD"
    }
  },
  "Manifests": [
```

```
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "install": "pip3 install --user boto3",
        "Run": "python3 -u {artifacts:path}/list_s3_buckets.py"
      }
    },
    {
      "Platform": {
        "os": "windows"
      },
      "Lifecycle": {
        "install": "pip3 install --user boto3",
        "Run": "py -3 -u {artifacts:path}/list_s3_buckets.py"
      }
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.ListS3Buckets
ComponentVersion: '1.0.0'
ComponentDescription: A component that uses the token exchange service to list S3
 buckets.
ComponentPublisher: Amazon
ComponentDependencies:
  aws.greengrass.TokenExchangeService:
    VersionRequirement: '^2.0.0'
    DependencyType: HARD
Manifests:
  - Platform:
      os: linux
    Lifecycle:
      install:
        pip3 install --user boto3
      Run: |-
        python3 -u {artifacts:path}/list_s3_buckets.py
  - Platform:
```

```
      os: windows
    Lifecycle:
      install:
        pip3 install --user boto3
      Run: |-
        py -3 -u {artifacts:path}/list_s3_buckets.py
```

This example component runs the following Python script, `list_s3_buckets.py` that lists Amazon S3 buckets.

```
import boto3
import os

try:
    print("Creating boto3 S3 client...")
    s3 = boto3.client('s3')
    print("Successfully created boto3 S3 client")
except Exception as e:
    print("Failed to create boto3 s3 client. Error: " + str(e))
    exit(1)

try:
    print("Listing S3 buckets...")
    response = s3.list_buckets()
    for bucket in response['Buckets']:
        print(f'\t{bucket["Name"]}')
    print("Successfully listed S3 buckets")
except Exception as e:
    print("Failed to list S3 buckets. Error: " + str(e))
    exit(1)
```

Run a Docker container

You can configure AWS IoT Greengrass components to run a Docker container from images stored in the following locations:

- Public and private image repositories in Amazon Elastic Container Registry (Amazon ECR)
- Public Docker Hub repository
- Public Docker Trusted Registry
- S3 bucket

In your custom component, include the Docker image URI as an artifact to retrieve the image and run it on the core device. For Amazon ECR and Docker Hub images, you can use the Docker application manager component to download the images and manage credentials for private Amazon ECR repositories.

Topics

- Requirements
- Run a Docker container from a public image in Amazon ECR or Docker Hub
- Run a Docker container from a private image in Amazon ECR
- Run a Docker container from an image in Amazon S3
- Use interprocess communication in Docker container components
- Use AWS credentials in Docker container components (Linux)
- Use stream manager in Docker container components (Linux)

Requirements

To run a Docker container in a component, you need the following:

- A Greengrass core device. If you don't have one, see Tutorial: Getting started with AWS IoT Greengrass V2.
- Docker Engine 1.9.1 or later installed on the Greengrass core device. Versions up to and including 29.6.0, such as 20.10, 24.0, 25.0, and 27.5, work with the AWS IoT Greengrass Core software. You must install Docker directly on the core device before you deploy components that run Docker containers.

> **Tip**
>
> You can also configure the core device to install Docker Engine when the component installs. For example, the following install script installs Docker Engine before it loads the Docker image. This install script works on Debian-based Linux distributions, such as Ubuntu. If you configure the component to install Docker Engine with this command, you may need to set `RequiresPrivilege` to `true` in the lifecycle script to run the installation and `docker` commands. For more information, see AWS IoT Greengrass component recipe reference.

```
apt-get install docker-ce docker-ce-cli containerd.io && docker load -i
 {artifacts:path}/hello-world.tar
```

- The system user that runs a Docker container component must have root or administrator permissions, or you must configure Docker to run it as a non-root or non-admistrator user.
  - On Linux devices, you can add a user to the `docker` group to call `docker` commands without `sudo`.
  - On Windows devices, you can add a user to the `docker-users` group to call `docker` commands without adminstrator privileges.

  Linux or Unix

  To add `ggc_user`, or the non-root user that you use to run Docker container components, to the `docker` group, run the following command.

  ```
  sudo usermod -aG docker ggc_user
  ```

  For more information, see Manage Docker as a non-root user.

  Windows Command Prompt (CMD)

  To add `ggc_user`, or the user that you use to run Docker container components, to the `docker-users` group, run the following command as an administrator.

  ```
  net localgroup docker-users ggc_user /add
  ```

  Windows PowerShell

  To add `ggc_user`, or the user that you use to run Docker container components, to the `docker-users` group, run the following command as an administrator.

  ```
  Add-LocalGroupMember -Group docker-users -Member ggc_user
  ```

- Files accessed by the Docker container component mounted as a volume in the Docker container.
- If you configure the AWS IoT Greengrass Core software to use a network proxy, you must configure Docker to use the same proxy server.

In addition to these requirements, you must also meet the following requirements if they apply to your environment:

- To use Docker Compose to create and start your Docker containers, install Docker Compose on your Greengrass core device, and upload your Docker Compose file to an S3 bucket. You must store your Compose file in an S3 bucket in the same AWS account and AWS Region as the component. For an example that uses the `docker-compose up` command in a custom component, see Run a Docker container from a public image in Amazon ECR or Docker Hub.
- If you run AWS IoT Greengrass behind a network proxy, configure the Docker daemon to use a proxy server.
- If your Docker images are stored in Amazon ECR or Docker Hub, include the Docker component manager component as a dependency in your Docker container component. You must start the Docker daemon on the core device before you deploy your component.

  Also, include the image URIs as component artifacts. Image URIs must be in the format `docker:`*`registry`*`/`*`image`*`[`*`:tag|@digest`*`]` as shown in the following examples:

  - Private Amazon ECR image: `docker:`*`account-id`*`.dkr.ecr.`*`region`*`.amazonaws.com/`*`repository`*`/`*`image`*`[`*`:tag|@digest`*`]`
  - Public Amazon ECR image: `docker:public.ecr.aws/`*`repository`*`/`*`image`*`[`*`:tag|@digest`*`]`
  - Public Docker Hub image: `docker:`*`name`*`[`*`:tag|@digest`*`]`

  For more information about running Docker containers from images stored in public repositories, see Run a Docker container from a public image in Amazon ECR or Docker Hub.
- If your Docker images are stored in an Amazon ECR private repository, then you must include the token exchange service component as a dependency in the Docker container component. Also, the Greengrass device role must allow the `ecr:GetAuthorizationToken`, `ecr:BatchGetImage`, and `ecr:GetDownloadUrlForLayer` actions, as shown in the following example IAM policy.

  JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Action": [
        "ecr:GetAuthorizationToken",
```

```
        "ecr:BatchGetImage",
        "ecr:GetDownloadUrlForLayer"
      ],
      "Resource": [
        "*"
      ],
      "Effect": "Allow"
    }
  ]
}
```

For information about running Docker containers from images stored in an Amazon ECR private repository, see Run a Docker container from a private image in Amazon ECR.

- To use Docker images stored in an Amazon ECR private repository, the private repository must be in the same AWS Region as the core device.
- If your Docker images or Compose files are stored in an S3 bucket, the Greengrass device role must allow the `s3:GetObject` permission to allow core devices to download the images as component artifacts, as shown in the following example IAM policy.

  JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Action": [
        "s3:GetObject"
      ],
      "Resource": [
        "*"
      ],
      "Effect": "Allow"
    }
  ]
}
```

For information about running Docker containers from images stored in Amazon S3, see Run a Docker container from an image in Amazon S3.

- To use interprocess communication (IPC), AWS credentials, or stream manager in your Docker container component, you must specify additional options when you run the Docker container. For more information, see the following:
  - Use interprocess communication in Docker container components
  - Use AWS credentials in Docker container components (Linux)
  - Use stream manager in Docker container components (Linux)

Run a Docker container from a public image in Amazon ECR or Docker Hub

This section describes how you can create a custom component that uses Docker Compose to run a Docker container from Docker images that are stored Amazon ECR and Docker Hub.

To run a Docker container using Docker Compose

1. Create and upload a Docker Compose file to an Amazon S3 bucket. Make sure that the Greengrass device role allows the `s3:GetObject` permission to enable the device to access the Compose file. The example Compose file shown in the following example includes the Amazon CloudWatch Agent image from Amazon ECR and the MySQL image from Docker Hub.

   ```
   version: "3"
   services:
     cloudwatchagent:
       image: "public.ecr.aws/cloudwatch-agent/cloudwatch-agent:latest"
     mysql:
       image: "mysql:8.0"
   ```

2. Create a custom component on your AWS IoT Greengrass core device. The example recipe shown in the following example has the following properties:

   - The Docker application manager component as a dependency. This component enables AWS IoT Greengrass to download images from public Amazon ECR and Docker Hub repositories.
   - A component artifact that specifies a Docker image in a public Amazon ECR repository.
   - A component artifact that specifies a Docker image in a public Docker Hub repository.
   - A component artifact that specifies the Docker Compose file that includes containers for the Docker images that you want to run.
   - A lifecycle run script that uses docker-compose up to create and start a container from the specified images.

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.MyDockerComposeComponent",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that uses Docker Compose to run images
 from public Amazon ECR and Docker Hub.",
  "ComponentPublisher": "Amazon",
  "ComponentDependencies": {
    "aws.greengrass.DockerApplicationManager": {
      "VersionRequirement": "~2.0.0"
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "all"
      },
      "Lifecycle": {
        "Run": "docker-compose -f {artifacts:path}/docker-compose.yaml up"
      },
      "Artifacts": [
        {
          "URI": "docker:public.ecr.aws/cloudwatch-agent/cloudwatch-
agent:latest"
        },
        {
          "URI": "docker:mysql:8.0"
        },
        {
          "URI": "s3://amzn-s3-demo-bucket/folder/docker-compose.yaml"
        }
      ]
    }
  ]
}
```

YAML

```
---
```

```
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.MyDockerComposeComponent
ComponentVersion: '1.0.0'
ComponentDescription: 'A component that uses Docker Compose to run images from
 public Amazon ECR and Docker Hub.'
ComponentPublisher: Amazon
ComponentDependencies:
  aws.greengrass.DockerApplicationManager:
    VersionRequirement: ~2.0.0
Manifests:
  - Platform:
      os: all
    Lifecycle:
        Run: docker-compose -f {artifacts:path}/docker-compose.yaml up
    Artifacts:
      - URI: "docker:public.ecr.aws/cloudwatch-agent/cloudwatch-agent:latest"
      - URI: "docker:mysql:8.0"
      - URI: "s3://amzn-s3-demo-bucket/folder/docker-compose.yaml"
```

Note

To use interprocess communication (IPC), AWS credentials, or stream manager in your Docker container component, you must specify additional options when you run the Docker container. For more information, see the following:

- Use interprocess communication in Docker container components
- Use AWS credentials in Docker container components (Linux)
- Use stream manager in Docker container components (Linux)

3. Test the component to verify that it works as expected.

Important

You must install and start the Docker daemon before you deploy the component.

After you deploy the component locally, you can run the docker container ls command to verify that your container runs.

```
docker container ls
```

4. When the component is ready, upload the component to AWS IoT Greengrass to deploy to other core devices. For more information, see Publish components to deploy to your core devices.

Run a Docker container from a private image in Amazon ECR

This section describes how you can create a custom component that runs a Docker container from a Docker image that is stored in a private repository in Amazon ECR.

To run a Docker container

1. Create a custom component on your AWS IoT Greengrass core device. Use the following example recipe, which has the following properties:

   - The Docker application manager component as a dependency. This component enables AWS IoT Greengrass to manage credentials to download images from private repositories.
   - The token exchange service component as a dependency. This component enables AWS IoT Greengrass to retrieve AWS credentials to interact with Amazon ECR.
   - A component artifact that specifies a Docker image in a private Amazon ECR repository.
   - A lifecycle run script that uses docker run to create and start a container from the image.

   JSON

   ```
   {
     "RecipeFormatVersion": "2020-01-25",
     "ComponentName": "com.example.MyPrivateDockerComponent",
     "ComponentVersion": "1.0.0",
     "ComponentDescription": "A component that runs a Docker container from a
    private Amazon ECR image.",
     "ComponentPublisher": "Amazon",
     "ComponentDependencies": {
       "aws.greengrass.DockerApplicationManager": {
         "VersionRequirement": "~2.0.0"
       },
       "aws.greengrass.TokenExchangeService": {
         "VersionRequirement": "~2.0.0"
   ```

```
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "all"
      },
      "Lifecycle": {
        "Run": "docker run account-
id.dkr.ecr.region.amazonaws.com/repository[:tag|@digest]"
      },
      "Artifacts": [
        {
          "URI": "docker:account-
id.dkr.ecr.region.amazonaws.com/repository[:tag|@digest]"
        }
      ]
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.MyPrivateDockerComponent
ComponentVersion: '1.0.0'
ComponentDescription: 'A component that runs a Docker container from a private
 Amazon ECR image.'
ComponentPublisher: Amazon
ComponentDependencies:
  aws.greengrass.DockerApplicationManager:
    VersionRequirement: ~2.0.0
  aws.greengrass.TokenExchangeService:
    VersionRequirement: ~2.0.0
Manifests:
  - Platform:
      os: all
    Lifecycle:
        Run: docker run account-id.dkr.ecr.region.amazonaws.com/repository[:tag|
@digest]
    Artifacts:
```

```
      - URI: "docker:account-id.dkr.ecr.region.amazonaws.com/repository[:tag|
@digest]"
```

> **Note**
>
> To use interprocess communication (IPC), AWS credentials, or stream manager in your Docker container component, you must specify additional options when you run the Docker container. For more information, see the following:
>
> - Use interprocess communication in Docker container components
> - Use AWS credentials in Docker container components (Linux)
> - Use stream manager in Docker container components (Linux)

2. Test the component to verify that it works as expected.

   > **Important**
   >
   > You must install and start the Docker daemon before you deploy the component.

   After you deploy the component locally, you can run the docker container ls command to verify that your container runs.

   ```
   docker container ls
   ```

3. Upload the component to AWS IoT Greengrass to deploy to other core devices. For more information, see Publish components to deploy to your core devices.

Run a Docker container from an image in Amazon S3

This section describes how you can run a Docker container in a component from a Docker image that is stored in Amazon S3.

To run a Docker container in a component from an image in Amazon S3

1. Run the docker save command to create a backup of a Docker container. You provide this backup as a component artifact to run the container on AWS IoT Greengrass. Replace *hello-*

world with the name of the image, and replace *hello-world.tar* with the name of the archive file to create.

```
docker save hello-world > artifacts/com.example.MyDockerComponent/1.0.0/hello-world.tar
```

2. Create a custom component on your AWS IoT Greengrass core device. Use the following example recipe, which has the following properties:

   - A lifecycle install script that uses docker load to load a Docker image from an archive.
   - A lifecycle run script that uses docker run to create and start a container from the image. The `--rm` option cleans up the container when it exits.

   JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.MyS3DockerComponent",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that runs a Docker container from an
 image in an S3 bucket.",
  "ComponentPublisher": "Amazon",
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "install": {
          "Script": "docker load -i {artifacts:path}/hello-world.tar"
        },
        "Run": {
          "Script": "docker run --rm hello-world"
        }
      }
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.MyS3DockerComponent
ComponentVersion: '1.0.0'
ComponentDescription: 'A component that runs a Docker container from an image in
 an S3 bucket.'
ComponentPublisher: Amazon
Manifests:
  - Platform:
      os: linux
    Lifecycle:
      install:
        Script: docker load -i {artifacts:path}/hello-world.tar
      Run:
        Script: docker run --rm hello-world
```

> **Note**
>
> To use interprocess communication (IPC), AWS credentials, or stream manager in your Docker container component, you must specify additional options when you run the Docker container. For more information, see the following:
>
> - Use interprocess communication in Docker container components
> - Use AWS credentials in Docker container components (Linux)
> - Use stream manager in Docker container components (Linux)

3. Test the component to verify that it works as expected.

   After you deploy the component locally, you can run the docker container ls command to verify that your container runs.

   ```
   docker container ls
   ```

4. When the component is ready, upload the Docker image archive to an S3 bucket, and add its URI to the component recipe. Then, you can upload the component to AWS IoT Greengrass to deploy to other core devices. For more information, see Publish components to deploy to your core devices.

When you're done, the component recipe should look like the following example.

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.MyS3DockerComponent",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that runs a Docker container from an
 image in an S3 bucket.",
  "ComponentPublisher": "Amazon",
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "install": {
          "Script": "docker load -i {artifacts:path}/hello-world.tar"
        },
        "Run": {
          "Script": "docker run --rm hello-world"
        }
      },
      "Artifacts": [
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.MyDockerComponent/1.0.0/hello-world.tar"
        }
      ]
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.MyS3DockerComponent
ComponentVersion: '1.0.0'
ComponentDescription: 'A component that runs a Docker container from an image in
 an S3 bucket.'
```

```
ComponentPublisher: Amazon
Manifests:
  - Platform:
      os: linux
    Lifecycle:
      install:
        Script: docker load -i {artifacts:path}/hello-world.tar
      Run:
        Script: docker run --rm hello-world
    Artifacts:
      - URI: s3://amzn-s3-demo-bucket/artifacts/
com.example.MyDockerComponent/1.0.0/hello-world.tar
```

Use interprocess communication in Docker container components

You can use the Greengrass interprocess communication (IPC) library in the AWS IoT Device SDK or the AWS IoT Greengrass Component SDK to communicate with the Greengrass nucleus, other Greengrass components, and AWS IoT Core. For more information, see Use the AWS IoT Device SDK to communicate with the Greengrass nucleus, other components, and AWS IoT Core.

To use IPC in a Docker container component, you must run the Docker container with the following parameters:

- Mount the IPC socket in the container. The Greengrass nucleus provides the IPC socket file path in the `AWS_GG_NUCLEUS_DOMAIN_SOCKET_FILEPATH_FOR_COMPONENT` environment variable.
- Set the `SVCUID` and `AWS_GG_NUCLEUS_DOMAIN_SOCKET_FILEPATH_FOR_COMPONENT` environment variables to the values that the Greengrass nucleus provides to components. Your component uses these environment variables to authenticate connections to the Greengrass nucleus.

Example recipe: Publish an MQTT message to AWS IoT Core (Python)

The following recipe defines an example Docker container component that publishes an MQTT message to AWS IoT Core. This recipe has the following properties:

- An authorization policy (`accessControl`) that allows the component to publish MQTT messages to AWS IoT Core on all topics. For more information, see Authorize components to perform IPC operations and AWS IoT Core MQTT IPC authorization.
- A component artifact that specifies a Docker image as a TAR archive in Amazon S3.

- A lifecycle install script that loads the Docker image from the TAR archive.
- A lifecycle run script that runs a Docker container from the image. The Docker run command has the following arguments:
  - The `-v` argument mounts the Greengrass IPC socket in the container.
  - The first two `-e` arguments set the required environment variables in the Docker container.
  - The additional `-e` arguments set environment variables used by this example.
  - The `--rm` argument cleans up the container when it exits.

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.python.docker.PublishToIoTCore",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "Uses interprocess communication to publish an MQTT
 message to IoT Core.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "topic": "test/topic/java",
      "message": "Hello, World!",
      "qos": "1",
      "accessControl": {
        "aws.greengrass.ipc.mqttproxy": {
          "com.example.python.docker.PublishToIoTCore:pubsub:1": {
            "policyDescription": "Allows access to publish to IoT Core on all
 topics.",
            "operations": [
              "aws.greengrass#PublishToIoTCore"
            ],
            "resources": [
              "*"
            ]
          }
        }
      }
    }
  },
  "Manifests": [
    {
```

```
      "Platform": {
        "os": "all"
      },
      "Lifecycle": {
        "install": "docker load -i {artifacts:path}/publish-to-iot-core.tar",
        "Run": "docker run -v $AWS_GG_NUCLEUS_DOMAIN_SOCKET_FILEPATH_FOR_COMPONENT:
$AWS_GG_NUCLEUS_DOMAIN_SOCKET_FILEPATH_FOR_COMPONENT -e SVCUID -e
 AWS_GG_NUCLEUS_DOMAIN_SOCKET_FILEPATH_FOR_COMPONENT -e MQTT_TOPIC=
\"{configuration:/topic}\" -e MQTT_MESSAGE=\"{configuration:/message}\" -e MQTT_QOS=
\"{configuration:/qos}\" --rm publish-to-iot-core"
      },
      "Artifacts": [
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.python.docker.PublishToIoTCore/1.0.0/publish-to-iot-core.tar"
        }
      ]
    }
  ]
}
```

YAML

```
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.python.docker.PublishToIoTCore
ComponentVersion: 1.0.0
ComponentDescription: Uses interprocess communication to publish an MQTT message to
 IoT Core.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    topic: 'test/topic/java'
    message: 'Hello, World!'
    qos: '1'
    accessControl:
      aws.greengrass.ipc.mqttproxy:
        'com.example.python.docker.PublishToIoTCore:pubsub:1':
          policyDescription: Allows access to publish to IoT Core on all topics.
          operations:
            - 'aws.greengrass#PublishToIoTCore'
          resources:
            - '*'
Manifests:
```

```
  - Platform:
      os: all
    Lifecycle:
      install: 'docker load -i {artifacts:path}/publish-to-iot-core.tar'
      Run: |
        docker run \
          -v $AWS_GG_NUCLEUS_DOMAIN_SOCKET_FILEPATH_FOR_COMPONENT:
$AWS_GG_NUCLEUS_DOMAIN_SOCKET_FILEPATH_FOR_COMPONENT \
          -e SVCUID \
          -e AWS_GG_NUCLEUS_DOMAIN_SOCKET_FILEPATH_FOR_COMPONENT \
          -e MQTT_TOPIC="{configuration:/topic}" \
          -e MQTT_MESSAGE="{configuration:/message}" \
          -e MQTT_QOS="{configuration:/qos}" \
          --rm publish-to-iot-core
    Artifacts:
      - URI: s3://amzn-s3-demo-bucket/artifacts/
com.example.python.docker.PublishToIoTCore/1.0.0/publish-to-iot-core.tar
```

Use AWS credentials in Docker container components (Linux)

You can use the token exchange service component to interact with AWS services in Greengrass components. This component provides AWS credentials from the core device's token exchange role using a local container server. For more information, see Interact with AWS services.

> **Note**
>
> The example in this section works only on Linux core devices.

To use AWS credentials from the token exchange service in a Docker container component, you must run the Docker container with the following parameters:

- Provide access to the host network using the `--network=host` argument. This option enables the Docker container to connect to the local token exchange service to retrieve AWS credentials. This argument works on only Docker for Linux.

  > **Warning**
  >
  > This option gives the container access to all local network interfaces on the host, so this option is less secure than if you run Docker containers without this access to the host

network. Consider this when you develop and run Docker container components that use this option. For more information, see Network: host in the *Docker Documentation*.

- Set the `AWS_CONTAINER_CREDENTIALS_FULL_URI` and `AWS_CONTAINER_AUTHORIZATION_TOKEN` environment variables to the values that the Greengrass nucleus provides to components. AWS SDKs use these environment variables to retrieve AWS credentials.

Example recipe: List S3 buckets in a Docker container component (Python)

The following recipe defines an example Docker container component that lists the S3 buckets in your AWS account. This recipe has the following properties:

- The token exchange service component as a dependency. This dependency enables the component to retrieve AWS credentials to interact with other AWS services.
- A component artifact that specifies a Docker image as a tar archive in Amazon S3.
- A lifecycle install script that loads the Docker image from the TAR archive.
- A lifecycle run script that runs a Docker container from the image. The Docker run command has the following arguments:
  - The `--network=host` argument provides the container access to the host network, so the container can connect to the token exchange service.
  - The `-e` argument sets the required environment variables in the Docker container.
  - The `--rm` argument cleans up the container when it exits.

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.python.docker.ListS3Buckets",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "Uses the token exchange service to lists your S3
 buckets.",
  "ComponentPublisher": "Amazon",
  "ComponentDependencies": {
    "aws.greengrass.TokenExchangeService": {
      "VersionRequirement": "^2.0.0",
      "DependencyType": "HARD"
```

```
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "install": "docker load -i {artifacts:path}/list-s3-buckets.tar",
        "Run": "docker run --network=host -e AWS_CONTAINER_AUTHORIZATION_TOKEN -e
 AWS_CONTAINER_CREDENTIALS_FULL_URI --rm list-s3-buckets"
      },
      "Artifacts": [
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.python.docker.ListS3Buckets/1.0.0/list-s3-buckets.tar"
        }
      ]
    }
  ]
}
```

YAML

```
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.python.docker.ListS3Buckets
ComponentVersion: 1.0.0
ComponentDescription: Uses the token exchange service to lists your S3 buckets.
ComponentPublisher: Amazon
ComponentDependencies:
  aws.greengrass.TokenExchangeService:
    VersionRequirement: ^2.0.0
    DependencyType: HARD
Manifests:
  - Platform:
      os: linux
    Lifecycle:
      install: 'docker load -i {artifacts:path}/list-s3-buckets.tar'
      Run: |
        docker run \
          --network=host \
          -e AWS_CONTAINER_AUTHORIZATION_TOKEN \
          -e AWS_CONTAINER_CREDENTIALS_FULL_URI \
```

```
            --rm list-s3-buckets
    Artifacts:
      - URI: s3://amzn-s3-demo-bucket/artifacts/
com.example.python.docker.ListS3Buckets/1.0.0/list-s3-buckets.tar
```

Use stream manager in Docker container components (Linux)

You can use the stream manager component to manage data streams in Greengrass components. This component enables you to process data streams and transfer high-volume IoT data to the AWS Cloud. AWS IoT Greengrass provides a stream manager SDK that you use to interact with the stream manager component. For more information, see Manage data streams on Greengrass core devices.

> **Note**
>
> The example in this section works only on Linux core devices.

To use the stream manager SDK in a Docker container component, you must run the Docker container with the following parameters:

- Provide access to the host network using the `--network=host` argument. This option enables the Docker container to interact with the stream manager component over a local TLS connection. This argument works on only Docker for Linux

  > **Warning**
  >
  > This option gives the container access to all local network interfaces on the host, so this option is less secure than if you run Docker containers without this access to the host network. Consider this when you develop and run Docker container components that use this option. For more information, see Network: host in the *Docker Documentation*.

- If you configure the stream manager component to require authentication, which is the default behavior, set the `AWS_CONTAINER_CREDENTIALS_FULL_URI` environment variable to the value that the Greengrass nucleus provides to components. For more information, see stream manager configuration.
- If you configure the stream manager component to use a non-default port, use interprocess communication (IPC) to get the port from the stream manager component configuration. You

must run the Docker container with additional options to use IPC. For more information, see the following:

- Connect to stream manager in application code
- Use interprocess communication in Docker container components

Example recipe: Stream a file to an S3 bucket in a Docker container component (Python)

The following recipe defines an example Docker container component that creates a file and streams it to an S3 bucket. This recipe has the following properties:

- The stream manager component as a dependency. This dependency enables the component to use the stream manager SDK to interact with the stream manager component.
- A component artifact that specifies a Docker image as a TAR archive in Amazon S3.
- A lifecycle install script that loads the Docker image from the TAR archive.
- A lifecycle run script that runs a Docker container from the image. The Docker run command has the following arguments:
  - The `--network=host` argument provides the container access to the host network, so the container can connect to the stream manager component.
  - The first `-e` argument sets the required `AWS_CONTAINER_AUTHORIZATION_TOKEN` environment variable in the Docker container.
  - The additional `-e` arguments set environment variables used by this example.
  - The `-v` argument mounts the component's work folder in the container. This example creates a file in the work folder to upload that file to Amazon S3 using stream manager.
  - The `--rm` argument cleans up the container when it exits.

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.python.docker.StreamFileToS3",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "Creates a text file and uses stream manager to stream the
 file to S3.",
  "ComponentPublisher": "Amazon",
  "ComponentDependencies": {
    "aws.greengrass.StreamManager": {
```

```
      "VersionRequirement": "^2.0.0",
      "DependencyType": "HARD"
    }
  },
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "bucketName": ""
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "install": "docker load -i {artifacts:path}/stream-file-to-s3.tar",
        "Run": "docker run --network=host -e AWS_CONTAINER_AUTHORIZATION_TOKEN
 -e BUCKET_NAME=\"{configuration:/bucketName}\" -e WORK_PATH=\"{work:path}\" -v
 {work:path}:{work:path} --rm stream-file-to-s3"
      },
      "Artifacts": [
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.python.docker.StreamFileToS3/1.0.0/stream-file-to-s3.tar"
        }
      ]
    }
  ]
}
```

YAML

```
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.python.docker.StreamFileToS3
ComponentVersion: 1.0.0
ComponentDescription: Creates a text file and uses stream manager to stream the file
 to S3.
ComponentPublisher: Amazon
ComponentDependencies:
  aws.greengrass.StreamManager:
    VersionRequirement: ^2.0.0
    DependencyType: HARD
ComponentConfiguration:
```

```
  DefaultConfiguration:
    bucketName: ''
Manifests:
  - Platform:
      os: linux
    Lifecycle:
      install: 'docker load -i {artifacts:path}/stream-file-to-s3.tar'
      Run: |
        docker run \
          --network=host \
          -e AWS_CONTAINER_AUTHORIZATION_TOKEN \
          -e BUCKET_NAME="{configuration:/bucketName}" \
          -e WORK_PATH="{work:path}" \
          -v {work:path}:{work:path} \
          --rm stream-file-to-s3
    Artifacts:
      - URI: s3://amzn-s3-demo-bucket/artifacts/
com.example.python.docker.StreamFileToS3/1.0.0/stream-file-to-s3.tar
```

AWS IoT Greengrass component recipe reference

The component recipe is a file that defines a component's details, dependencies, artifacts, and lifecycles. The component *lifecycle* specifies the commands to run to install, run, and shut down the component, for example. The AWS IoT Greengrass core uses the lifecycles that you define in the recipe to install and run components. The AWS IoT Greengrass service uses the recipe to identify the dependencies and artifacts to deploy to your core devices when you deploy the component.

In the recipe, you can define unique dependencies and lifecycles for each platform that a component supports. You can use this capability to deploy a component to devices with multiple platforms that have different requirements. You can also use this to prevent AWS IoT Greengrass from installing a component on devices that don't support it.

Each recipe contains a list of *manifests*. Each manifest specifies a set of platform requirements and the lifecycle and artifacts to use for core devices whose platform meets those requirements. The core device uses the first manifest with platform requirements that the device meets. Specify a manifest without any platform requirements to match any core device.

You can also specify a global lifecycle that isn't in a manifest. In the global lifecycle, you can use *selection keys* that identify sub-sections of the lifecycle. Then, you can specify these selection keys within a manifest to use those sections of the global lifecycle in addition to the manifest's

lifecycle. The core device uses the manifest's selection keys only if the manifest doesn't define a lifecycle. You can use the `all` selection in a manifest to match sections of the global lifecycle without selection keys.

After the AWS IoT Greengrass Core software selects a manifest that matches the core device, it does the following to identify the lifecycle steps to use:

- If the selected manifest defines a lifecycle, the core device uses that lifecycle.
- If the selected manifest doesn't define a lifecycle, the core device uses the global lifecycle. The core device does the following to identify which sections of the global lifecycle to use:
  - If the manifest defines selection keys, the core device uses the sections of the global lifecycle that contain the manifest's selection keys.
  - If the manifest doesn't define selection keys, the core device uses the sections of the global lifecycle that don't have selection keys. This behavior is equivalent to a manifest that defines the `all` selection.

> **Important**
>
> A core device must match least one manifest's platform requirements to install the component. If no manifest matches the core device, then the AWS IoT Greengrass Core software doesn't install the component and the deployment fails.

You can define recipes in JSON or YAML format. The recipe examples section includes recipes in each format.

Topics

- Recipe validation
- Recipe format
- Recipe variables
- Recipe examples

Recipe validation

Greengrass validates a JSON or YAML component recipe when creating a component version. This recipe validation checks your JSON or YAML component recipe for common errors in order

to prevent potential deployment issues. The validation checks the recipe for common errors (e.g., missing commas, braces, and fields) and to make sure the recipe is well-formed.

If you receive a recipe validation error message, check your recipe for any missing commas, braces, or fields. Verify that you are not missing any fields by looking at the recipe format.

Recipe format

When you define a recipe for a component, you specify the following information in the recipe document. The same structure applies to recipes in YAML and JSON formats.

`RecipeFormatVersion`

The template version for the recipe. Choose the following option:

- `2020-01-25`

`ComponentName`

The name of the component that this recipe defines. The component name must be unique in your AWS account in each Region.

Tips

- Use inverse domain name format to avoid name collision within your company. For example, if your company owns `example.com` and you work on a solar energy project, you can name your Hello World component `com.example.solar.HelloWorld`. This helps avoid component name collisions within your company.
- Avoid the `aws.greengrass` prefix in your component names. AWS IoT Greengrass uses this prefix for the public components that it provides. If you choose the same name as a public component, your component replaces that component. Then, AWS IoT Greengrass provides your component instead of the public component when it deploys components with a dependency on that public component. This feature enables you to override the behavior of public components, but it can also break other components if you don't intend to override a public component.

`ComponentVersion`

The version of the component. The maximum value for the major, minor, and patch values is 999999.

> **Note**
>
> AWS IoT Greengrass uses semantic versions for components. Semantic versions follow a *major.minor.patch* number system. For example, version `1.0.0` represents the first major release for a component. For more information, see the semantic version specification.

`ComponentDescription`

(Optional) The description of the component.

`ComponentPublisher`

The publisher or author of the component.

`ComponentConfiguration`

(Optional) An object that defines the configuration or parameters for the component. You define the default configuration, and then when you deploy the component, you can specify the configuration object to provide to the component. Component configuration supports nested parameters and structures. This object contains the following information:

`DefaultConfiguration`

An object that defines the default configuration for the component. You define the structure of this object.

> **Note**
>
> AWS IoT Greengrass uses JSON for configuration values. JSON specifies a number type but doesn't differentiate between integers and floats. As a result, configuration values might convert to floats in AWS IoT Greengrass. To ensure that your component uses the correct data type, we recommend that you define numeric configuration values as strings. Then, have your component parse them as integers or floats. This ensures that your configuration values have the same type in the configuration and on your core device.

`ComponentDependencies`

(Optional) A dictionary of objects that each define a component dependency for the component. The key for each object identifies the name of the component dependency. AWS IoT Greengrass installs component dependencies when the component installs. AWS IoT Greengrass waits for dependencies to start before it starts the component. Each object contains the following information:

`VersionRequirement`

The npm-style semantic version constraint that defines the compatible component versions for this dependency. You can specify a version or a range of versions. For more information, see the npm semantic version calculator.

`DependencyType`

(Optional) The type of this dependency. Choose from the following options.

- `SOFT` – The component doesn't restart if the dependency changes state.
- `HARD` – The component restarts if the dependency changes state.

Defaults to `HARD`.

`ComponentType`

(Optional) The type of component.

> **Note**
>
> We don't recommend that you specify the component type in a recipe. AWS IoT Greengrass sets the type for you when you create a component.

The type can be one the following types:

- `aws.greengrass.generic` – The component runs commands or provides artifacts.
- `aws.greengrass.lambda` – The component runs a Lambda function using the Lambda launcher component. The `ComponentSource` parameter specifies the ARN of the Lambda function that this component runs.

  We don't recommend that you use this option, because it's set by AWS IoT Greengrass when you create a component from a Lambda function. For more information, see Run AWS Lambda functions.

- `aws.greengrass.plugin` – The component runs in the same Java Virtual Machine (JVM) as the Greengrass nucleus. If you deploy or restart a plugin component, the Greengrass nucleus restarts.

  Plugin components use the same log file as the Greengrass nucleus. For more information, see Monitor AWS IoT Greengrass logs.

  We don't recommend that you use this option in component recipes, because it's intended for AWS-provided components written in Java that directly interface with the Greengrass nucleus. For more information about which public components are plugins, see AWS-provided components.
- `aws.greengrass.nucleus` – The nucleus component. For more information, see Greengrass nucleus.

  We don't recommend that you use this option in component recipes. It is intended for the Greengrass nucleus component, which provides the minimum functionality of the AWS IoT Greengrass Core software.

Defaults to `aws.greengrass.generic` when you create a component from a recipe, or `aws.greengrass.lambda` when you create a component from a Lambda function.

For more information, see Component types.

`ComponentSource`

(Optional) The ARN of the Lambda function that a component runs.

We don't recommend that you specify the component source in a recipe. AWS IoT Greengrass sets this parameter for you when you create a component from a Lambda function. For more information, see Run AWS Lambda functions.

`Manifests`

A list of objects that each define the component's lifecycle, parameters, and requirements for a platform. If a core device matches multiple manifests' platform requirements, AWS IoT Greengrass uses the first manifest that the core device matches. To ensure that core devices use the correct manifest, define the manifests with stricter platform requirements first. A manifest that applies to all platforms must be the last manifest in the list.

> **Important**
>
> A core device must match least one manifest's platform requirements to install the component. If no manifest matches the core device, then the AWS IoT Greengrass Core software doesn't install the component and the deployment fails.

Each object contains the following information:

`Name`

(Optional) A friendly name for the platform that this manifest defines.

If you omit this parameter, AWS IoT Greengrass creates a name from the platform `os` and `architecture`.

`Platform`

(Optional) An object that defines the platform to which this manifest applies. Omit this parameter to define a manifest that applies to all platforms.

This object specifies key-value pairs about the platform on which a core device runs. When you deploy this component, the AWS IoT Greengrass Core software compares these key-value pairs with the platform attributes on the core device. The AWS IoT Greengrass Core software always defines `os` and `architecture`, and it might define additional attributes. You can specify custom platform attributes for a core device when you deploy the Greengrass nucleus component. For more information, see the platform overrides parameter of the Greengrass nucleus component.

For each key-value pair, you can specify one of the following values:

- An exact value, such as `linux` or `windows`. Exact values must start with a letter or a number.
- `*`, which matches any value. This also matches when a value isn't present.
- A Java-style regular expression, such as `/windows|linux/`. The regular expression must start and end with a slash character (`/`). For example, the regular expression `/.+/` matches any non-blank value.

This object contains the following information:

`runtime`

The Greengrass nucleus runtime for the platform that this manifest supports. When defining multiple manifests with platform `runtime`, The supported runtime values in a recipe are `aws_nucleus_lite` and `*` only. To target a classic device, runtime field MUST NOT be specified in the the recipe. Supported Greengrass Nucleus runtimes include the following values:

- `*`
- `aws_nucleus_lite`

`os`

(Optional) The name of the operating system for the platform that this manifest supports. Common platforms include the following values:

- `linux`
- `windows`
- `darwin` (macOS)

`architecture`

(Optional) The processor architecture for the platform that this manifest supports. Common architectures include the following values:

- `amd64`
- `arm`
- `aarch64`
- `x86`

`architecture.detail`

(Optional) The processor architecture detail for the platform that this manifest supports. Common architecture details include the following values:

- `arm61`
- `arm71`
- `arm81`

`key`

(Optional) A platform attribute that you define for this manifest. Replace *`Key`* with the name of the platform attribute. The AWS IoT Greengrass Core software matches this

platform attribute with the key-value pairs that you specify in the Greengrass nucleus component configuration. For more information, see the platform overrides parameter of the Greengrass nucleus component.

> **Tip**
>
> Use inverse domain name format to avoid name collision within your company. For example, if your company owns `example.com` and you work on a radio project, you can name a custom platform attribute `com.example.radio.RadioModule`. This helps avoid platform attribute name collisions within your company.

For example, you might define a platform attribute, `com.example.radio.RadioModule`, to specify a different manifest based on which radio module is available on a core device. Each manifest can include different artifacts that apply to different hardware configurations, so that you deploy the minimal set of software to the core device.

`Lifecycle`

An object or string that defines how to install and run the component on the platform that this manifest defines. You can also define a global lifecycle that applies to all platforms. The core device uses the global lifecycle only if the manifest to use doesn't specify a lifecycle.

> **Note**
>
> You define this lifecycle within a manifest. The lifecycle steps that you specify here apply to only the platform that this manifest defines. You can also define a global lifecycle that applies to all platforms.

This object or string contains the following information:

`Setenv`

(Optional) A dictionary of environment variables to provide to all lifecycle scripts. You can override these environment variables with `Setenv` in each lifecycle script.

`install`

(Optional) An object or string that defines the script to run when the component installs. The AWS IoT Greengrass Core software also runs this lifecycle step each time the software launches.

If the `install` script exits with a success code, the component enters the `INSTALLED` state.

This object or string contains the following information:

`Script`

The script to run.

`RequiresPrivilege`

(Optional) If you set this option to `true`, the AWS IoT Greengrass Core software runs this lifecycle script as the user that runs the Greengrass nucleus instead of the system user configured for this component. When the nucleus runs as root, this grants root privileges to the script. Defaults to `false`.

`Skipif`

(Optional) The check to determine whether or not to run the script. You can define to check if an executable is on the path or if a file exists. If the output is true, then the AWS IoT Greengrass Core software skips the step. Choose one of the following checks:

- `onpath` *`runnable`* – Check if a runnable is on the system path. For example, use **`onpath python3`** to skip this lifecycle step if Python 3 is available.
- `exists` *`file`* – Check if a file exists. For example, use **`exists /tmp/my-configuration.db`** to skip this lifecycle step if `/tmp/my-configuration.db` is present.

`Timeout`

(Optional) The maximum amount of time in seconds that the script can run before the AWS IoT Greengrass Core software terminates the process.

Default: 120 seconds

`Setenv`

(Optional) The dictionary of environment variables to provide to the script. These environment variables override the variables that you provide in `Lifecycle.Setenv`.

`run`

(Optional) An object or string that defines the script to run when the component starts.

The component enters the `RUNNING` state when this lifecycle step runs. If the `run` script exits with a success code, the component enters the `STOPPING` state. If a `shutdown` script is specified, it runs; otherwise the component enters the `FINISHED` state.

Components that depend on this component start when this lifecycle step runs. To run a background process, such as a service that dependent components use, use the `startup` lifecycle step instead.

When you deploy components with a `run` lifecycle, the core device can report the deployment as complete as soon as this lifecycle script runs. As a result, the deployment can be complete and successful even if the `run` lifecycle script fails soon after running. If you want the deployment status to depend on the result of the component's start script, use the `startup` lifecycle step instead.

> **Note**
>
> You can define only one `startup` or `run` lifecycle.

This object or string contains the following information:

`Script`

The script to run.

`RequiresPrivilege`

(Optional) If you set this option to `true`, the AWS IoT Greengrass Core software runs this lifecycle script as the user that runs the Greengrass nucleus instead of the system user configured for this component. When the nucleus runs as root, this grants root privileges to the script. Defaults to `false`.

`Skipif`

(Optional) The check to determine whether or not to run the script. You can define to check if an executable is on the path or if a file exists. If the output is true, then the AWS IoT Greengrass Core software skips the step. Choose one of the following checks:

- `onpath` *`runnable`* – Check if a runnable is on the system path. For example, use **`onpath python3`** to skip this lifecycle step if Python 3 is available.
- `exists` *`file`* – Check if a file exists. For example, use **`exists /tmp/my-configuration.db`** to skip this lifecycle step if `/tmp/my-configuration.db` is present.

`Timeout`

(Optional) The maximum amount of time in seconds that the script can run before the AWS IoT Greengrass Core software terminates the process.

This lifecycle step doesn't timeout by default. If you omit this timeout, the `run` script runs until it exits.

`Setenv`

(Optional) The dictionary of environment variables to provide to the script. These environment variables override the variables that you provide in `Lifecycle.Setenv`.

`startup`

(Optional) An object or string that defines the background process to run when the component starts.

Use `startup` to run a command that must exit successfully or update the component's status to `RUNNING` before dependent components can start. Use the UpdateState IPC operation to set the component's status to `RUNNING` or `ERRORED` when the component starts a script that doesn't exit. For example, you might define a `startup` step that starts the MySQL process with `/etc/init.d/mysqld start`.

The component enters the `STARTING` state when this lifecycle step runs. If the `startup` script exits with a success code, the component enters the `RUNNING` state. Then, dependent components can start.

When you deploy components with a `startup` lifecycle, the core device can report the deployment as complete after this lifecycle script exits or reports its state. In other

words, the deployment's status is `IN_PROGRESS` until all components' startup scripts exit or report a state.

> **Note**
>
> You can define only one `startup` or `run` lifecycle.

This object or string contains the following information:

`Script`

The script to run.

`RequiresPrivilege`

(Optional) If you set this option to `true`, the AWS IoT Greengrass Core software runs this lifecycle script as the user that runs the Greengrass nucleus instead of the system user configured for this component. When the nucleus runs as root, this grants root privileges to the script. Defaults to `false`.

`Skipif`

(Optional) The check to determine whether or not to run the script. You can define to check if an executable is on the path or if a file exists. If the output is true, then the AWS IoT Greengrass Core software skips the step. Choose one of the following checks:

- `onpath` *`runnable`* – Check if a runnable is on the system path. For example, use **`onpath python3`** to skip this lifecycle step if Python 3 is available.
- `exists` *`file`* – Check if a file exists. For example, use **`exists /tmp/my-configuration.db`** to skip this lifecycle step if `/tmp/my-configuration.db` is present.

`Timeout`

(Optional) The maximum amount of time in seconds that the script can run before the AWS IoT Greengrass Core software terminates the process.

Default: 120 seconds

`Setenv`

(Optional) The dictionary of environment variables to provide to the script. These environment variables override the variables that you provide in `Lifecycle.Setenv`.

`shutdown`

(Optional) An object or string that defines the script to run when the component shuts down. Use the shutdown lifecycle to execute code that you want to run when the component is in the `STOPPING` state. The shutdown lifecycle can be used to stop a process started by the `startup` or `run` scripts.

If you start a background process in `startup`, use the `shutdown` step to stop that process when the component shuts down. For example, you might define a `shutdown` step that stops the MySQL process with `/etc/init.d/mysqld stop`.

The `shutdown` script runs after the component enters the `STOPPING` state. If the script completes successfully, the component enters the `FINISHED` state.

This object or string contains the following information:

`Script`

The script to run.

`RequiresPrivilege`

(Optional) If you set this option to `true`, the AWS IoT Greengrass Core software runs this lifecycle script as the user that runs the Greengrass nucleus instead of the system user configured for this component. When the nucleus runs as root, this grants root privileges to the script. Defaults to `false`.

`Skipif`

(Optional) The check to determine whether or not to run the script. You can define to check if an executable is on the path or if a file exists. If the output is true, then the AWS IoT Greengrass Core software skips the step. Choose one of the following checks:

- `onpath` *`runnable`* – Check if a runnable is on the system path. For example, use **`onpath python3`** to skip this lifecycle step if Python 3 is available.
- `exists` *`file`* – Check if a file exists. For example, use **`exists /tmp/my-configuration.db`** to skip this lifecycle step if `/tmp/my-configuration.db` is present.

`Timeout`

(Optional) The maximum amount of time in seconds that the script can run before the AWS IoT Greengrass Core software terminates the process.

Default: 15 seconds.

`Setenv`

(Optional) The dictionary of environment variables to provide to the script. These environment variables override the variables that you provide in `Lifecycle.Setenv`.

`uninstall`

(Optional) An object or string that defines the script to run when the component is permanently removed from the device. The AWS IoT Greengrass Core software runs this step when a deployment removes a component. This step does not run during component version upgrades.

> **Note**
>
> This feature requires Greengrass nucleus version 2.17.0 or later. On earlier nucleus versions, the AWS IoT Greengrass Core software does not run the uninstall lifecycle step when it removes a component.

The component enters the `UNINSTALLING` state when this lifecycle step runs. If the uninstall script fails or times out, the component is still removed. Uninstall failures do not fail the deployment.

This object or string contains the following information:

`Script`

The script to run.

`RequiresPrivilege`

(Optional) If you set this option to `true`, the AWS IoT Greengrass Core software runs this lifecycle script as the user that runs the Greengrass nucleus instead of the system user configured for this component. When the nucleus runs as root, this grants root privileges to the script. Defaults to `false`.

`Skipif`

(Optional) The check to determine whether or not to run the script. You can define to check if an executable is on the path or if a file exists. If the output is true, then the AWS IoT Greengrass Core software skips the step. Choose one of the following checks:

- `onpath` *`runnable`* – Check if a runnable is on the system path. For example, use **`onpath python3`** to skip this lifecycle step if Python 3 is available.
- `exists` *`file`* – Check if a file exists. For example, use **`exists /tmp/my-configuration.db`** to skip this lifecycle step if `/tmp/my-configuration.db` is present.

`Timeout`

(Optional) The maximum amount of time in seconds that the script can run before the AWS IoT Greengrass Core software terminates the process.

Default: 120 seconds

`Setenv`

(Optional) The dictionary of environment variables to provide to the script. These environment variables override the variables that you provide in `Lifecycle.Setenv`.

`recover`

(Optional) An object or string that defines the script to run when the component encounters an error.

This step runs when a component enters the `ERRORED` state. If the component becomes `ERRORED` three times without successfully recovering, the component changes to the `BROKEN` state. To fix a `BROKEN` component, you must deploy it again.

This object or string contains the following information:

`Script`

The script to run.

`RequiresPrivilege`

(Optional) If you set this option to `true`, the AWS IoT Greengrass Core software runs this lifecycle script as the user that runs the Greengrass nucleus instead of the system

user configured for this component. When the nucleus runs as root, this grants root privileges to the script. Defaults to `false`.

`Skipif`

(Optional) The check to determine whether or not to run the script. You can define to check if an executable is on the path or if a file exists. If the output is true, then the AWS IoT Greengrass Core software skips the step. Choose one of the following checks:

- `onpath` *`runnable`* – Check if a runnable is on the system path. For example, use **`onpath python3`** to skip this lifecycle step if Python 3 is available.
- `exists` *`file`* – Check if a file exists. For example, use **`exists /tmp/my-configuration.db`** to skip this lifecycle step if `/tmp/my-configuration.db` is present.

`Timeout`

(Optional) The maximum amount of time in seconds that the script can run before the AWS IoT Greengrass Core software terminates the process.

Default: 60 seconds.

`Setenv`

(Optional) The dictionary of environment variables to provide to the script. These environment variables override the variables that you provide in `Lifecycle.Setenv`.

`bootstrap`

(Optional) An object or string that defines a script that requires the AWS IoT Greengrass Core software or core device to restart. This lets you develop a component that performs a restart after it installs operating system updates or runtime updates, for example.

> **Note**
>
> To install updates or dependencies that don't require the AWS IoT Greengrass Core software or device to restart, use the install lifecycle.

This lifecycle step runs before the install lifecycle step in the following cases when the AWS IoT Greengrass Core software deploys the component:

- The component deploys to the core device for the first time.

- The component version changes.
- The bootstrap script changes as the result of a component configuration update.

After the AWS IoT Greengrass Core software completes the bootstrap step for all components that have a bootstrap step in a deployment, the software restarts.

> **Note**
>
> The AWS IoT Greengrass Core software ignores the failure handling policy for bootstrap deployments. If a deployment that runs bootstrap lifecycle steps fails, the core device rolls back to its previous configuration. The rollback happens even if you set the failure handling policy to `DO_NOTHING`. For more information, see Create deployments.

> **Important**
>
> You must configure the AWS IoT Greengrass Core software as a system service to restart the AWS IoT Greengrass Core software or the core device. If you don't configure the AWS IoT Greengrass Core software as a system service, the software won't restart. For more information, see Configure the Greengrass nucleus as a system service.

This object or string contains the following information:

`BootstrapOnRollback`

> **Note**
>
> When this feature is enabled, `BootstrapOnRollback` will only run for components that have either completed or attempted to run the bootstrap lifecycle steps as part of a failed target deployment. This feature is available for Greengrass nucleus versions 2.12.0 and later.

(Optional) You can run the bootstrap lifecycle steps as part of a rollback deployment. If you set this option to `true`, the bootstrap lifecycle steps defined within a rollback

deployment will run. When a deployment fails, the previous version of the component bootstrap lifecycle will run again during a rollback deployment.

Defaults to `false`.

`Script`

The script to run. The exit code of this script defines the restart instruction. Use the following exit codes:

- `0` – Don't restart the AWS IoT Greengrass Core software or the core device. The AWS IoT Greengrass Core software still restarts after all components bootstrap.
- `100` – Request to restart the AWS IoT Greengrass Core software.
- `101` – Request to restart the core device.

Exit codes 100 to 199 are reserved for special behavior. Other exit codes represent script errors.

`RequiresPrivilege`

(Optional) If you set this option to `true`, the AWS IoT Greengrass Core software runs this lifecycle script as the user that runs the Greengrass nucleus instead of the system user configured for this component. When the nucleus runs as root, this grants root privileges to the script. Defaults to `false`.

`Timeout`

(Optional) The maximum amount of time in seconds that the script can run before the AWS IoT Greengrass Core software terminates the process.

Default: 120 seconds

`Setenv`

(Optional) The dictionary of environment variables to provide to the script. These environment variables override the variables that you provide in `Lifecycle.Setenv`.

`Selections`

(Optional) A list of selection keys that specify sections of the global lifecycle to run for this manifest. In the global lifecycle, you can define lifecycle steps with selection keys at any level to select sub-sections of the lifecycle. Then, the core device uses those sections that

match the selection keys in this manifest. For more information, see the global lifecycle examples.

> **Important**
>
> The core device uses the selections from the global lifecycle only if this manifest doesn't define a lifecycle.

You can specify the `all` selection key to run sections of the global lifecycle that don't have selection keys.

`Artifacts`

(Optional) A list of objects that each define a binary artifact for the component on the platform that this manifest defines. For example, you can define code or images as artifacts.

When the component deploys, the AWS IoT Greengrass Core software downloads the artifact to a folder on the core device. You can also define artifacts as archive files that the software extracts after it downloads them.

You can use recipe variables to get the paths to the folders where the artifacts install on the core device.

- Normal files – Use the artifacts:path recipe variable to get the path to the folder that contains the artifacts. For example, specify `{artifacts:path}/my_script.py` in a recipe to get the path to an artifact that has the URI `s3://amzn-s3-demo-bucket/path/to/my_script.py`.
- Extracted archives – Use the artifacts:decompressedPath recipe variable to get the path to the folder that contains the extracted archive artifacts. The AWS IoT Greengrass Core software extracts each archive to a folder with the same name as the archive. For example, specify `{artifacts:decompressedPath}/my_archive/my_script.py` in a recipe to get the path to `my_script.py` in the archive artifact that has the URI `s3://amzn-s3-demo-bucket/path/to/my_archive.zip`.

> **Note**
>
> When you develop a component with an archive artifact on a local core device, you might not have a URI for that artifact. To test your component with an `Unarchive`

option that extracts the artifact, specify a URI where the file name matches the name of your archive artifact file. You can specify the URI where you expect to upload the archive artifact, or you can specify a new placeholder URI. For example, to extract the `my_archive.zip` artifact during a local deployment, you can specify `s3://amzn-s3-demo-bucket/my_archive.zip`.

Each object contains the following information:

`Uri`

The URI of an artifact in an S3 bucket. The AWS IoT Greengrass Core software fetches the artifact from this URI when the component installs, unless the artifact already exists on the device. Each artifact must have a unique file name within each manifest.

`Unarchive`

(Optional) The type of archive to unpack. Choose from the following options:

- `NONE` – The file isn't an archive to unpack. The AWS IoT Greengrass Core software installs the artifact to a folder on the core device. You can use the artifacts:path recipe variable to get the path to this folder.
- `ZIP` – The file is a ZIP archive. The AWS IoT Greengrass Core software extracts the archive to a folder with the same name as the archive. You can use the artifacts:decompressedPath recipe variable to get the path to the folder that contains this folder.

Defaults to `NONE`.

`Permission`

(Optional) An object that defines the access permissions to set for this artifact file. You can set the read permission and the execute permission.

Note

You can't set the write permission, because the AWS IoT Greengrass Core software doesn't allow components to edit artifact files in the artifacts folder. To edit an artifact file in a component, copy it to another location or publish and deploy a new artifact file.

If you define an artifact as an archive to unpack, then the AWS IoT Greengrass Core software sets these access permissions on the files that it unpacks from the archive. The AWS IoT Greengrass Core software sets the folder's access permissions to `ALL` for `Read` and `Execute`. This allows components to view the unpacked files in the folder. To set permissions on individual files from the archive, you can set the permissions in the install lifecycle script.

This object contains the following information:

`Read`

(Optional) The read permission to set for this artifact file. To allow other components to access this artifact, such as components that depend on this component, specify `ALL`. Choose from the following options:

- `NONE` – The file isn't readable.
- `OWNER` – The file is readable by the system user that you configure to run this component.
- `ALL` – The file is readable by all users.

Defaults to `OWNER`.

`Execute`

(Optional) The run permission to set for this artifact file. The `Execute` permission implies the `Read` permission. For example, if you specify `ALL` for `Execute`, then all users can read and run this artifact file.

Choose from the following options:

- `NONE` – The file isn't runnable.
- `OWNER` – The file is runnable by the system user that you configure to run the component.
- `ALL` – The file is runnable by all users.

Defaults to `NONE`.

`Digest`

(Read-only) The cryptographic digest hash of the artifact. When you create a component, AWS IoT Greengrass uses a hash algorithm to calculate a hash of the artifact file. Then,

when you deploy the component, the Greengrass nucleus calculates the hash of the downloaded artifact and compares the hash with this digest to verify the artifact before installation. If the hash doesn't match the digest, the deployment fails.

If you set this parameter, AWS IoT Greengrass replaces the value that you set when you create the component.

`Algorithm`

(Read-only) The hash algorithm that AWS IoT Greengrass uses to calculate the digest hash of the artifact.

If you set this parameter, AWS IoT Greengrass replaces the value that you set when you create the component.

`Lifecycle`

An object that defines how to install and run the component. The core device uses the global lifecycle only if the manifest to use doesn't specify a lifecycle.

> **Note**
>
> You define this lifecycle outside a manifest. You can also define a manifest lifecycle that applies to the platforms that match that manifest.

In the global lifecycle, you can specify lifecycles that run for certain selection keys that you specify in each manifest. Selection keys are strings that identify sections of the global lifecycle to run for each manifest.

The `all` selection key is the default on any section without a selection key. This means that you can specify the `all` selection key in a manifest to run the sections of the global lifecycle without selection keys. You don't need to specify the `all` selection key in the global lifecycle.

If a manifest doesn't define a lifecycle or selection keys, the core device defaults to use the `all` selection. This means that in this case, the core device uses the sections of the global lifecycle that don't use selection keys.

This object contains the same information as the manifest lifecycle, but you can specify selection keys at any level to select sub-sections of the lifecycle.

> **Tip**
>
> We recommend that you use only lowercase letters for each selection key to avoid conflicts between selection keys and lifecycle keys. Lifecycle keys start with a capital letter.

Example global lifecycle with top-level selection keys

```
Lifecycle:
  key1:
    install:
      SkipIf: either onpath executable or exists file
      Script: command1
  key2:
    install:
      Script: command2
  all:
    install:
      Script: command3
```

Example global lifecycle with bottom-level selection keys

```
Lifecycle:
  install:
    Script:
      key1: command1
      key2: command2
      all: command3
```

Example global lifecycle with multiple levels of selection keys

```
Lifecycle:
  key1:
    install:
      SkipIf: either onpath executable or exists file
      Script: command1
  key2:
    install:
      Script: command2
  all:
```

```
    install:
      Script:
        key3: command3
        key4: command4
        all: command5
```

Recipe variables

Recipe variables expose information from the current component and nucleus for you to use in your recipes. For example, you can use a recipe variable to pass component configuration parameters to an application that you run in a lifecycle script.

You can use recipe variables in the following sections of component recipes:

- Lifecycle definitions.
- Component configuration definitions, if you use Greengrass nucleus v2.6.0 or later and set the interpolateComponentConfiguration configuration option to `true`. You can also use recipes variables when you deploy component configuration updates.

Recipe variables use `{recipe_variable}` syntax. The curly braces indicate a recipe variable.

AWS IoT Greengrass supports the following recipe variables:

`component_dependency_name``:configuration:`*`json_pointer`*

The value of a configuration parameter for the component that this recipe defines or for a component on which this component depends.

You can use this variable to provide a parameter to a script that you run in the component lifecycle.

> **Note**
>
> AWS IoT Greengrass supports this recipe variable only in component lifecycle definitions.

This recipe variable has the following inputs:

- `component_dependency_name` – (Optional) The name of the component dependency to query. Omit this segment to query the component that this recipe defines. You can specify only direct dependencies.
- `json_pointer` – The JSON pointer to the configuration value to evaluate. JSON pointers start with a forward slash `/`. To identify a value in a nested component configuration, use forward slashes (`/`) to separate the keys for each level in the configuration. You can use a number as a key to specify an index in a list. For more information, see the JSON pointer specification.

  AWS IoT Greengrass Core uses JSON pointers for recipes in YAML format.

The JSON pointer can reference the following node types:

- A value node. AWS IoT Greengrass Core replaces the recipe variable with the string representation of the value. Null values convert to `null` as a string.
- An object node. AWS IoT Greengrass Core replaces the recipe variable with the serialized JSON string representation of that object.
- No node. AWS IoT Greengrass Core doesn't replace the recipe variable.

For example, the `{configuration:/Message}` recipe variable retrieves the value of the `Message` key in the component configuration. The `{com.example.MyComponentDependency:configuration:/server/port}` recipe variable retrieves the value of `port` in the `server` configuration object of a component dependency.

`component_dependency_name``:artifacts:path`

The root path of the artifacts for the component that this recipe defines or for a component on which this component depends.

When a component installs, AWS IoT Greengrass copies the component's artifacts to the folder that this variable exposes. You can use this variable to identify the location of a script to run in the component lifecycle, for example.

The folder at this path is read-only. To modify artifact files, copy the files to another location, such as the current working directory (`$PWD` or `.`). Then, modify the files there.

To read or run an artifact from a component dependency, that artifact's `Read` or `Execute` permission must be `ALL`. For more information, see the artifact permissions that you define in the component recipe.

This recipe variable has the following inputs:

- `component_dependency_name` – (Optional) The name of the component dependency to query. Omit this segment to query the component that this recipe defines. You can specify only direct dependencies.

`component_dependency_name``:artifacts:decompressedPath`

The root path of the decompressed archive artifacts for the component that this recipe defines or for a component on which this component depends.

When a component installs, AWS IoT Greengrass unpacks the component's archive artifacts to the folder that this variable exposes. You can use this variable to identify the location of a script to run in the component lifecycle, for example.

Each artifact unzips to a folder within the decompressed path, where the folder has the same name as the artifact minus its extension. For example, a ZIP artifact named `models.zip` unpacks to the `{artifacts:decompressedPath}/models` folder.

The folder at this path is read-only. To modify artifact files, copy the files to another location, such as the current working directory (`$PWD` or `.`). Then, modify the files there.

To read or run an artifact from a component dependency, that artifact's `Read` or `Execute` permission must be `ALL`. For more information, see the artifact permissions that you define in the component recipe.

This recipe variable has the following inputs:

- `component_dependency_name` – (Optional) The name of the component dependency to query. Omit this segment to query the component that this recipe defines. You can specify only direct dependencies.

`component_dependency_name``:work:path`

This feature is available for v2.0.4 and later of the Greengrass nucleus component.

The work path for the component that this recipe defines or for a component on which this component depends. The value of this recipe variable is equivalent to the output of the `$PWD` environment variable and the pwd command when run from the context of the component.

You can use this recipe variable to share files between a component and a dependency.

The folder at this path is readable and writable by the component that this recipe defines and by other components that run as the same user and group.

This recipe variable has the following inputs:

- `component_dependency_name` – (Optional) The name of the component dependency to query. Omit this segment to query the component that this recipe defines. You can specify only direct dependencies.

`kernel:rootPath`

The AWS IoT Greengrass Core root path.

`iot:thingName`

This feature is available for v2.3.0 and later of the Greengrass nucleus component.

The name of the core device's AWS IoT thing.

Recipe examples

You can reference the following recipe examples to help you create recipes for your components.

AWS IoT Greengrass curates an index of Greengrass components, called the Greengrass Software Catalog. This catalog tracks Greengrass components that are developed by the Greengrass community. From this catalog, you can download, modify, and deploy components to create your Greengrass applications. For more information, see Community components.

Topics

- Hello World component recipe
- Python runtime component example
- Component recipe that specifies several fields

Hello World component recipe

The following recipe describes a Hello World component that runs a Python script. This component supports all platforms and accepts a `Message` parameter that AWS IoT Greengrass passes as an argument to the Python script. This is the recipe for the Hello World component in the Getting started tutorial.

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
```

```
  "ComponentName": "com.example.HelloWorld",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "My first AWS IoT Greengrass component.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "Message": "world"
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "run": "python3 -u {artifacts:path}/hello_world.py {configuration:/Message}"
      }
    },
    {
      "Platform": {
        "os": "windows"
      },
      "Lifecycle": {
        "run": "py -3 -u {artifacts:path}/hello_world.py {configuration:/Message}"
      }
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.HelloWorld
ComponentVersion: '1.0.0'
ComponentDescription: My first AWS IoT Greengrass component.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    Message: world
Manifests:
  - Platform:
      os: linux
```

```
    Lifecycle:
      run: |
        python3 -u {artifacts:path}/hello_world.py "{configuration:/Message}"
  - Platform:
      os: windows
    Lifecycle:
      run: |
        py -3 -u {artifacts:path}/hello_world.py "{configuration:/Message}"
```

Python runtime component example

The following recipe describes a component that installs Python. This component supports 64-bit Linux devices.

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.PythonRuntime",
  "ComponentDescription": "Installs Python 3.7",
  "ComponentPublisher": "Amazon",
  "ComponentVersion": "3.7.0",
  "Manifests": [
    {
      "Platform": {
        "os": "linux",
        "architecture": "amd64"
      },
      "Lifecycle": {
        "install": "apt-get update\napt-get install python3.7"
      }
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.PythonRuntime
ComponentDescription: Installs Python 3.7
ComponentPublisher: Amazon
```

```
ComponentVersion: '3.7.0'
Manifests:
  - Platform:
      os: linux
      architecture: amd64
    Lifecycle:
      install: |
        apt-get update
        apt-get install python3.7
```

Component recipe that specifies several fields

The following component recipe uses several recipe fields.

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.FooService",
  "ComponentDescription": "Complete recipe for AWS IoT Greengrass components",
  "ComponentPublisher": "Amazon",
  "ComponentVersion": "1.0.0",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "TestParam": "TestValue"
    }
  },
  "ComponentDependencies": {
    "BarService": {
      "VersionRequirement": "^1.1.0",
      "DependencyType": "SOFT"
    },
    "BazService": {
      "VersionRequirement": "^2.0.0"
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux",
        "architecture": "amd64"
      },
      "Lifecycle": {
```

```
        "install": {
          "Skipif": "onpath git",
          "Script": "sudo apt-get install git"
        },
        "Setenv": {
          "environment_variable1": "variable_value1",
          "environment_variable2": "variable_value2"
        }
      },
      "Artifacts": [
        {
          "Uri": "s3://amzn-s3-demo-bucket/hello_world.zip",
          "Unarchive": "ZIP"
        },
        {
          "Uri": "s3://amzn-s3-demo-bucket/hello_world_linux.py"
        }
      ]
    },
    {
      "Lifecycle": {
        "install": {
          "Skipif": "onpath git",
          "Script": "sudo apt-get install git",
          "RequiresPrivilege": "true"
        }
      },
      "Artifacts": [
        {
          "Uri": "s3://amzn-s3-demo-bucket/hello_world.py"
        }
      ]
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.FooService
ComponentDescription: Complete recipe for AWS IoT Greengrass components
ComponentPublisher: Amazon
```

```
ComponentVersion: 1.0.0
ComponentConfiguration:
  DefaultConfiguration:
    TestParam: TestValue
ComponentDependencies:
  BarService:
    VersionRequirement: ^1.1.0
    DependencyType: SOFT
  BazService:
    VersionRequirement: ^2.0.0
Manifests:
  - Platform:
      os: linux
      architecture: amd64
    Lifecycle:
      install:
        SkipIf: onpath git
        Script: sudo apt-get install git
      SetEnv:
        environment_variable1: variable_value1
        environment_variable2: variable_value2
    Artifacts:
      - Uri: 's3://amzn-s3-demo-bucket/hello_world.zip'
        Unarchive: ZIP
      - Uri: 's3://amzn-s3-demo-bucket/hello_world_linux.py'
  - Lifecycle:
      install:
        SkipIf: onpath git
        Script: sudo apt-get install git
        RequiresPrivilege: 'true'
    Artifacts:
      - Uri: 's3://amzn-s3-demo-bucket/hello_world.py'
```

Component environment variable reference

The AWS IoT Greengrass Core software sets environment variables when it runs lifecycle scripts for components. You can get these environment variables in your components to get the thing name, AWS Region, and Greengrass nucleus version. The software also sets environment variables that your component requires to use the interprocess communication SDK and to interact with AWS services.

You can also set custom environment variables for your component's lifecycle scripts. For more information, see Setenv.

The AWS IoT Greengrass Core software sets the following environment variables:

`AWS_IOT_THING_NAME`

The name of the AWS IoT thing that represents this Greengrass core device.

`AWS_REGION`

The AWS Region where this Greengrass core device operates.

The AWS SDKs use this environment variable to identify the default Region to use. This variable is equivalent to `AWS_DEFAULT_REGION`.

`AWS_DEFAULT_REGION`

The AWS Region where this Greengrass core device operates.

The AWS CLI uses this environment variable to identify the default Region to use. This variable is equivalent to `AWS_REGION`.

`GGC_VERSION`

The version of the Greengrass nucleus component that runs on this Greengrass core device.

`GG_ROOT_CA_PATH`

This feature is available for v2.5.5 and later of the Greengrass nucleus component.

The path to the root certificate authority (CA) certificate that the Greengrass nucleus uses.

`AWS_GG_NUCLEUS_DOMAIN_SOCKET_FILEPATH_FOR_COMPONENT`

The path to the IPC socket that components use to communicate with the AWS IoT Greengrass Core software. For more information, see Use the AWS IoT Device SDK to communicate with the Greengrass nucleus, other components, and AWS IoT Core.

`SVCUID`

The secret token that components use to connect to the IPC socket and communicate with the AWS IoT Greengrass Core software. For more information, see Use the AWS IoT Device SDK to communicate with the Greengrass nucleus, other components, and AWS IoT Core.

`AWS_CONTAINER_AUTHORIZATION_TOKEN`

The secret token that components use to retrieve credentials from the token exchange service component.

`AWS_CONTAINER_CREDENTIALS_FULL_URI`

The URI that components request to retrieve credentials from the token exchange service component.

Deploy AWS IoT Greengrass components to devices

You can use AWS IoT Greengrass to deploy components to devices or groups of devices. You use *deployments* to define the components and configurations that are sent to the devices. AWS IoT Greengrass deploys to *targets*, AWS IoT things or thing groups that represent Greengrass core devices. AWS IoT Greengrass uses AWS IoT Core jobs to deploy to your core devices. You can configure how the job rolls out to your devices.

Core device deployments

Each core device runs the components of the deployments for that device. A new deployment to the same target overwrites the previous deployment to the target. When you create a deployment, you define the components and configurations to apply to the core device's existing software.

When you revise a deployment for a target, you replace the components from the previous revision with the components in the new revision. For example, you deploy the Log manager and Secret manager components to the thing group `TestGroup`. Then you create another deployment for `TestGroup` that specifies only the secret manager component. As a result, the core devices in that group no longer run the log manager.

Platform dependency resolution

When a core device receives a deployment, it checks to make sure that the components are compatible with the core device. For example, if you deploy the Firehose to a Windows target, the deployment will fail.

Component dependency resolution

During a component deployment, the core device verifies compatibility of all components' dependencies and version requirements across a thing group. This verification ensures that version

contraints are satisfied for all components and their dependencies before proceeding with the deployment.

The dependency resolution process begins with identifying target components that have no dependencies in their recipes. Then, the system constructs a dependency tree using breadth-first search (BFS) which systematically explores each target node and finds their dependencies first before moving on to the next node. Each node includes the target component as the key and the version requirements as the value.

The version requirements combine three sets of constraints:

- The version requirements that are already established in the existing thing group.
- The component version required by the deployment. You must select a component version when you make or update a deployment.
- Any component version constraints that are defined within the recipe's dependency section.

Resolve component dependencies

During a deployment, the Greengrass nucleus first attempts to find the local candidate currently running on the device that satisfies the requirements. If the running component satisfies the requirements, the nucleus gets the stored recipe path from the recipe folder and finds the latest local version in the local store.

For AWS Cloud deployments, the nucleus will then call the ResolveComponentCandidates API. This API will start with the latest available version and check if it satisfies the dependencies and requirements. When the nucleus gets the response from the API, it selects that latest version. If there is no version found from the AWS Cloud that satisfies the requirements, the deployment fails. If the device is offline, it falls back to the original local candidate found. If there is no local candidate found that satisfies the requirements, the deployment fails.

For local deployments, the nucleus exclusively uses local candidates if they exist and if they satisfy the requirements without negotiating to AWS Cloud. If there is no such candidate, the deployment fails.

> **Note**
>
> All resolved recipes are stored locally for future reference.

For more information, see the dependency resolution section in GitHub.

If the Greengrass nucleus is able to successfully resolve all components, the nucleus log will contain the following line.

```
resolve-all-group-dependencies-finish. Finish resolving all groups dependencies.
```

Example

The following is an example of how the nucleus will resolve the component requirements.

- You deploy ComponentA which depends on ComponentC versions 1.0.0-1.9.0.
- You also deploy ComponentB which depends on ComponentC versions 1.4.0-1.9.5.

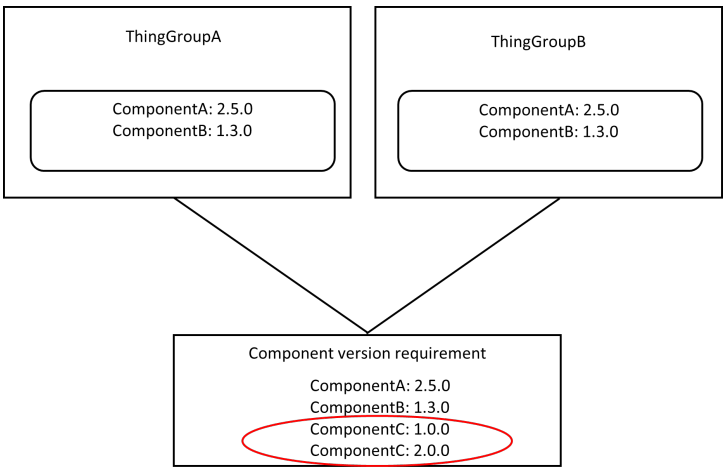
With component dependency resolution, the nucleus will deploy the latest version of ComponentC version to satisfy the requirements of ComponentA and ComponentB. This latest version of ComponentC is version 1.9.0.

Common component dependency resolution failures

The component dependency resolution may fail for two main reasons: target version requirement conflict or component dependency requirement conflict.

Scenario 1: Target version requirement conflict

- A thing exists in one thing group and you also want to add that thing to a new thing group. The deployment will fail if the new thing group requires a different thing version.
- A deployment may also fail if a thing belongs to a thing group and wants to update the component version through a thing deployment.



Failure log sample:

```
2025-04-11T06:16:03.315Z [ERROR] (pool-3-thread-27)
 com.aws.greengrass.componentmanager.ComponentManager: Failed to negotiate version
 with cloud and no local version to fall back to. {componentName=ComponentC,
 versionRequirement={thing/ABC==2.0.0, thinggroup/ThingGroupA==1.0.0}}
2025-04-11T06:16:03.316Z [ERROR] (pool-3-thread-26)
 com.aws.greengrass.deployment.DeploymentService: Error occurred while
 processing deployment. {deploymentId=fbac24de-4ef9-44b0-a685-fdc63b0f02b8,
 serviceName=DeploymentService, currentState=RUNNING}
java.util.concurrent.ExecutionException:
 com.aws.greengrass.componentmanager.exceptions.NoAvailableComponentVersionException:
 No local or cloud component version satisfies the requirements Check whether the
 version constraints conflict and that the component exists in your AWS account with
 a version that matches the version constraints. If the version constraints conflict,
 revise deployments to resolve the conflict. Component ComponentC version constraints:
 thing/ABC requires =2.0.0, thinggroup/ThingGroupA requires =1.0.0.
    at java.base/java.util.concurrent.FutureTask.report(FutureTask.java:122)
    at java.base/java.util.concurrent.FutureTask.get(FutureTask.java:191)
    at
 com.aws.greengrass.deployment.DefaultDeploymentTask.call(DefaultDeploymentTask.java:127)
    at
 com.aws.greengrass.deployment.DefaultDeploymentTask.call(DefaultDeploymentTask.java:50)
    at java.base/java.util.concurrent.FutureTask.run(FutureTask.java:264)
    at java.base/
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1128)
    at java.base/java.util.concurrent.ThreadPoolExecutor
$Worker.run(ThreadPoolExecutor.java:628)
    at java.base/java.lang.Thread.run(Thread.java:829)
Caused by:
 com.aws.greengrass.componentmanager.exceptions.NoAvailableComponentVersionException:
```

```
 No local or cloud component version satisfies the requirements Check whether the
 version constraints conflict and that the component exists in your AWS account with
 a version that matches the version constraints. If the version constraints conflict,
 revise deployments to resolve the conflict. Component ComponentC version constraints:
 thing/ABC requires =2.0.0, thinggroup/ThingGroupA requires =1.0.0.
    at
 com.aws.greengrass.componentmanager.ComponentManager.negotiateVersionWithCloud(ComponentManage
    at
 com.aws.greengrass.componentmanager.ComponentManager.resolveComponentVersion(ComponentManager.
    at com.aws.greengrass.componentmanager.DependencyResolver.lambda
$resolveDependencies$0(DependencyResolver.java:134)
    at
 com.aws.greengrass.componentmanager.DependencyResolver.resolveComponentDependencies(Dependency
    at
 com.aws.greengrass.componentmanager.DependencyResolver.resolveDependencies(DependencyResolver.
    at com.aws.greengrass.deployment.DefaultDeploymentTask.lambda$call
$2(DefaultDeploymentTask.java:125)
    ... 4 more
```

The logs indicate a version conflict error because the nucleus can't find a component version that simultaneously meetings two conflicting requirements.

How to resolve it

- If you want to keep the component in each thing group, select the same version of that component in each thing group.
- Select a component version that meets the deployment requirement.
- If you want to use a component version that doesn't meet both thing group requirements, select the component version that meets the thing group version requirement and use that component only in that thing group.

Scenario 2: Component dependency version requirement conflict

If a component is a dependency of different components and the components require different versions or different version ranges of that component, there a possibility that there are no available versions to satisfy all version requirements. In this scenario, the deployment will fail.

Example

Deployment of ComponentA (v2.5.0), ComponentB (v1.3.0), and ComponentC (v1.0.0)

- ComponentA requires ComponentB version >=1.0.0.

```
---
...
ComponentName: ComponentA
ComponentVersion: "2.5.0"
ComponentDependencies:
    ComponentB:
        VersionRequirement: ">=1.0.0"
        DependencyType: "HARD"
...
```

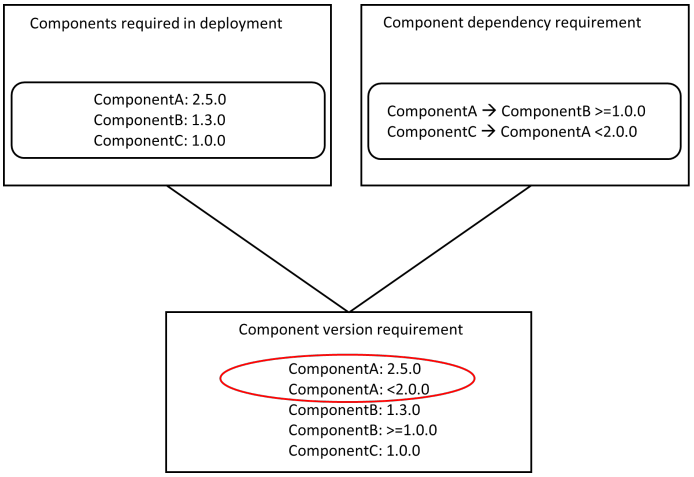
- ComponentC requires ComponentA version <2.0.0.

```
---
...
ComponentName: ComponentC
ComponentVersion: "1.0.0"
ComponentDependencies:
    ComponentA:
        VersionRequirement: "<2.0.0"
        DependencyType: "HARD"
...
```

There's a version conflict between two requirements for ComponentA:

- ComponentA requires version 2.5.0 in this deployment
- ComponentC requires ComponentA versions lower than 2.0.0

These two requirements contradict each other, making it impossible for the nucleus to find a ComponentA version that satisfies both requirements. Therefore, the dependency resolution fails.



Failure log sample:

```
2025-04-11T06:07:18.291Z [ERROR] (pool-3-thread-25)
 com.aws.greengrass.componentmanager.ComponentManager: Failed to negotiate version
 with cloud and no local version to fall back to. {componentName=ComponentA,
 versionRequirement={ComponentC=<2.0.0, thinggroup/ThingGroupA==2.5.0}}
2025-04-11T06:07:18.292Z [ERROR] (pool-3-thread-24)
 com.aws.greengrass.deployment.DeploymentService: Error occurred while
 processing deployment. {deploymentId=2ffac4df-1ac9-405c-8c11-28494a1b4382,
 serviceName=DeploymentService, currentState=RUNNING}
java.util.concurrent.ExecutionException:
 com.aws.greengrass.componentmanager.exceptions.NoAvailableComponentVersionException:
 No local or cloud component version satisfies the requirements Check whether the
 version constraints conflict and that the component exists in your AWS account with
 a version that matches the version constraints. If the version constraints conflict,
 revise deployments to resolve the conflict. Component ComponentA version constraints:
 ComponentC requires <2.0.0, thinggroup/ThingGroupA requires =2.5.0.
    at java.base/java.util.concurrent.FutureTask.report(FutureTask.java:122)
    at java.base/java.util.concurrent.FutureTask.get(FutureTask.java:191)
    at
 com.aws.greengrass.deployment.DefaultDeploymentTask.call(DefaultDeploymentTask.java:127)
    at
 com.aws.greengrass.deployment.DefaultDeploymentTask.call(DefaultDeploymentTask.java:50)
    at java.base/java.util.concurrent.FutureTask.run(FutureTask.java:264)
    at java.base/
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1128)
    at java.base/java.util.concurrent.ThreadPoolExecutor
$Worker.run(ThreadPoolExecutor.java:628)
    at java.base/java.lang.Thread.run(Thread.java:829)
Caused by:
 com.aws.greengrass.componentmanager.exceptions.NoAvailableComponentVersionException:
```

```
 No local or cloud component version satisfies the requirements Check whether the
 version constraints conflict and that the component exists in your AWS account with
 a version that matches the version constraints. If the version constraints conflict,
 revise deployments to resolve the conflict. Component ComponentA version constraints:
 ComponentC requires <2.0.0, thinggroup/ThingGroupA requires =2.5.0.
    at
 com.aws.greengrass.componentmanager.ComponentManager.negotiateVersionWithCloud(ComponentManage
    at
 com.aws.greengrass.componentmanager.ComponentManager.resolveComponentVersion(ComponentManager.
    at com.aws.greengrass.componentmanager.DependencyResolver.lambda
$resolveDependencies$0(DependencyResolver.java:134)
    at
 com.aws.greengrass.componentmanager.DependencyResolver.resolveComponentDependencies(Dependency
    at
 com.aws.greengrass.componentmanager.DependencyResolver.resolveDependencies(DependencyResolver.
    at com.aws.greengrass.deployment.DefaultDeploymentTask.lambda$call
$2(DefaultDeploymentTask.java:125)
    ... 4 more
```

The logs indicate that the nucleus can't find a version of ComponentA that satisfies the following requirements.

- The requirements for ComponentA to be exactly version 2.5.0 (from ThingGroupA).
- The requirement to work with ComponentC versions below 2.0.0.

How to resolve it

- If you want to keep the component in each thing group, select the same version of that component in each thing group.
- Select a component version that meets the deployment requirement.
- If you want to use a component version that doesn't meet both thing group requirements, select the component version that meets the thing group version requirement and use that component only in that thing group.

> **Tip**
>
> If you see this error on any AWS provided component, you can resolve it by updating the conflicted components to the latest version.

Removing a device from a thing group

When you remove a core device from a thing group, the component deployment behavior depends on the version of the Greengrass nucleus that the core device runs.

2.5.1 and later

When you remove a core device from a thing group, the behavior depends on whether the AWS IoT policy grants the `greengrass:ListThingGroupsForCoreDevice` permission. For more information about this permission and AWS IoT policies for core devices, see Device authentication and authorization for AWS IoT Greengrass.

- **If the AWS IoT policy grants this permission**

  When you remove a core device from a thing group, AWS IoT Greengrass removes the thing group's components the next time a deployment is made to the device. If a component on the device is included in the next deployment, that component is not removed from the device.

- **If the AWS IoT policy doesn't grant this permission**

  When you remove a core device from a thing group, AWS IoT Greengrass doesn't delete that thing group's components from the device.

  To remove a component from a device, use the deployment create command of the Greengrass CLI. Specify the component to remove with the `--remove` argument, and specify the thing group with the `--groupId` argument.

2.5.0

When you remove a core device from a thing group, AWS IoT Greengrass removes the thing group's components the next time a deployment is made to the device. If a component on the device is included in the next deployment, that component is not removed from the device.

This behavior requires that the core device's AWS IoT policy grants the `greengrass:ListThingGroupsForCoreDevice` permission. If a core device doesn't have this permission, the core device fails to apply deployments. For more information, see Device authentication and authorization for AWS IoT Greengrass.

2.0.x - 2.4.x

When you remove a core device from a thing group, AWS IoT Greengrass doesn't delete that thing group's components from the device.

To remove a component from a device, use the deployment create command of the Greengrass CLI. Specify the component to remove with the `--remove` argument, and specify the thing group with the `--groupId` argument.

Deployments

Deployments are continuous. When you create a deployment, AWS IoT Greengrass rolls out the deployment to target devices that are online. If a target device isn't online, then it receives the deployment the next time it connects to AWS IoT Greengrass. When you add a core device to a target thing group, AWS IoT Greengrass sends the device the latest deployment for that thing group.

Before a core device deploys a component, by default it notifies each component on the device. Greengrass components can respond to the notification to defer deployment. You might want to defer deployment if the device has a low battery level or is running a process that can't be interrupted. For more information, see Tutorial: Develop a Greengrass component that defers component updates. When you create a deployment you can configure it to deploy without notifying components.

Each target thing or thing group can have one deployment at a time. This means that when you create a deployment for a target, AWS IoT Greengrass no longer deploys the previous revision of that target's deployment.

Deployment options

Deployments provide several options that let you control which devices receive an update and how the update deploys. When you create a deployment, you can configure the following options:

- **AWS IoT Greengrass components**

  Define the components to install and run on the target devices. AWS IoT Greengrass components are software modules that you deploy and run on Greengrass core devices. Devices receive components only if the component supports the device's platform. This lets you deploy to

groups of devices even if the target devices run on multiple platforms. If a component doesn't support the device's platform, the component doesn't deploy to the device.

You can deploy custom components and AWS-provided components to your devices. When you deploy a component, AWS IoT Greengrass identifies any component dependencies and deploys them too. For more information, see Develop AWS IoT Greengrass components and AWS-provided components.

You define the version and configuration update to deploy for each component. The *configuration update* specifies how to modify the component's existing configuration on the core device, or the component's default configuration if the component doesn't exist on the core device. You can specify which configuration values to reset to default values and the new configuration values to merge onto the core device. When a core device receives deployments for different targets, and each deployment specifies compatible component versions, the core device applies configuration updates in order based on the timestamp of when you create the deployment. For more information, see Update component configurations.

> **Important**
>
> When you deploy a component, AWS IoT Greengrass installs the latest supported versions of all of that component's dependencies. Because of this, new patch versions of AWS-provided public components might be automatically deployed to your core devices if you add new devices to a thing group, or you update the deployment that targets those devices. Some automatic updates, such as a nucleus update, can cause your devices to restart unexpectedly.
>
> To prevent unintended updates for a component that is running on your device, we recommend that you directly include your preferred version of that component when you create a deployment. For more information about update behavior for AWS IoT Greengrass Core software, see Update the AWS IoT Greengrass Core software (OTA).

- **Deployment policies**

  Define when it's safe to deploy a configuration and what to do if the deployment fails. You can specify whether or not to wait for components to report that they can update. You can also specify whether or not to roll back devices to their previous configuration if they apply a deployment that fails.

- **Stop configuration**

Define when and how to stop a deployment. The deployment stops and fails if the criteria that you define are met. For example, you can configure a deployment to stop if a percentage of devices fail to apply that deployment after a minimum number of devices receive it.

- **Rollout configuration**

  Define the rate at which a deployments rolls out to the target devices. You can configure an exponential rate increase with minimum and maximum rate bounds.

- **Timeout configuration**

  Define the maximum amount of time each device has to apply a deployment. If a device exceeds the duration that you specify, then the device fails to apply the deployment.

> **Important**
>
> Custom components can define artifacts in S3 buckets. When the AWS IoT Greengrass Core software deploys a component, it downloads the component's artifacts from the AWS Cloud. Core device roles don't allow access to S3 buckets by default. To deploy custom components that define artifacts in an S3 bucket, the core device role must grant permissions to download artifacts from that bucket. For more information, see Allow access to S3 buckets for component artifacts.

Topics

- Create deployments
- Create subdeployments
- Revise deployments
- Cancel deployments
- Check deployment status

Create deployments

You can create a deployment that targets a thing or thing group.

When you create a deployment, you configure the software components to deploy and how the deployment job rolls out to target devices. You can define the deployment in the JSON file that you provide to the AWS CLI.

The deployment target determines the devices on which you want to run your components. To deploy to one core device, specify a thing. To deploy to multiple core devices, specify a thing group that includes those devices. For more information about how to configure thing groups, see Static thing groups and Dynamic thing groups in the *AWS IoT Developer Guide*.

Follow the steps in this section to create a deployment to a target. For more information about how to update the software components on a target that has a deployment, see Revise deployments.

> **Warning**
>
> The CreateDeployment operation can uninstall components from core devices. If a component is present in the previous deployment and not the new deployment, the core device uninstalls that component. To avoid uninstalling components, first use the ListDeployments operation to check if the target for the deployment already has an existing deployment. Then, use the GetDeployment operation to start from that existing deployment when you create a new deployment.

To create a deployment (AWS CLI)

1. Create a file called `deployment.json`, and then copy the following JSON object into the file. Replace *`targetArn`* with the ARN of the AWS IoT thing or thing group to target for the deployment. Thing and thing group ARNs have the following format:

   - Thing: `arn:aws:iot:`*`region`*`:`*`account-id`*`:thing/`*`thingName`*
   - Thing group: `arn:aws:iot:`*`region`*`:`*`account-id`*`:thinggroup/`*`thingGroupName`*

   ```
   {
     "targetArn": "targetArn"
   }
   ```

2. Check if the deployment target has an existing deployment that you want to revise. Do the following:

a. Run the following command to list the deployments for the deployment target. Replace *targetArn* with the ARN of the target AWS IoT thing or thing group.

```
aws greengrassv2 list-deployments --target-arn targetArn
```

The response contains a list with the latest deployment for the target. If the response is empty, the target doesn't have an existing deployment, and you can skip to Step 3. Otherwise, copy the `deploymentId` from the response to use in the next step.

> **Note**
>
> You can also revise a deployment other than the latest revision for the target. Specify the `--history-filter ALL` argument to list all deployments for the target. Then, copy the ID of the deployment that you want to revise.

b. Run the following command to get the deployment's details. These details include metadata, components, and job configuration. Replace *deploymentId* with the ID from the previous step.

```
aws greengrassv2 get-deployment --deployment-id deploymentId
```

The response contains the deployment's details.

c. Copy any of the following key-value pairs from the previous command's response into `deployment.json`. You can change these values for the new deployment.

- `deploymentName` – The deployment's name.
- `components` – The deployment's components. To uninstall a component, remove it from this object.
- `deploymentPolicies` – The deployment's policies.
- `iotJobConfiguration` – The deployment's job configuration.
- `tags` – The deployment's tags.

3. (Optional) Define a name for the deployment. Replace *deploymentName* with the name of the deployment.

```
{
  "targetArn": "targetArn",
```

```
  "deploymentName": "deploymentName"
}
```

4. Add each component to deploy the target devices. To do so, add key-value pairs to the `components` object, where the key is the component name, and the value is an object that contains the details for that component. Specify the following details for each component that you add:

   - `version` – The component version to deploy.
   - `configurationUpdate` – The configuration update to deploy. The update is a patch operation that modifies the component's existing configuration on each target device, or the component's default configuration if it doesn't exist on the target device. You can specify the following configuration updates:
     - Reset updates (`reset`) – (Optional) A list of JSON pointers that define the configuration values to reset to their default values on the target device. The AWS IoT Greengrass Core software applies reset updates before it applies merge updates. For more information, see Reset updates.
     - Merge updates (`merge`) – (Optional) A JSON document that defines the configuration values to merge onto the target device. You must serialize the JSON document as a string. For more information, see Merge updates.
   - `runWith` – (Optional) The system process options that the AWS IoT Greengrass Core software uses to run this component's processes on the core device. If you omit a parameter in the `runWith` object, the AWS IoT Greengrass Core software uses the default values that you configure on the Greengrass nucleus component.

     You can specify any of the following options:

     - `posixUser` – The POSIX system user and, optionally, group to use to run this component on Linux core devices. The user, and group if specified, must exist on each Linux core device. Specify the user and group separated by a colon (`:`) in the following format: `user:group`. The group is optional. If you don't specify a group, the AWS IoT Greengrass Core software uses the primary group for the user. For more information, see Configure the user that runs components.
     - `windowsUser` – The Windows user to use to run this component on Windows core devices. The user must exist on each Windows core device, and its name and password must be stored in the LocalSystem account's Credentials Manager instance. For more information, see Configure the user that runs components.

This feature is available for v2.5.0 and later of the Greengrass nucleus component.

- `systemResourceLimits` – The system resource limits to apply to this component's processes. You can apply system resource limits to generic and non-containerized Lambda components. For more information, see Configure system resource limits for components.

  You can specify any of the following options:

  - `cpus` – The maximum amount of CPU time that this component's processes can use on the core device. A core device's total CPU time is equivalent to the device's number of CPU cores. For example, on a core device with 4 CPU cores, you can set this value to 2 to limit this component's processes to 50 percent usage of each CPU core. On a device with 1 CPU core, you can set this value to `0.25` to limit this component's processes to 25 percent usage of the CPU. If you set this value to a number greater than the number of CPU cores, the AWS IoT Greengrass Core software doesn't limit the component's CPU usage.
  - `memory` – The maximum amount of RAM (in kilobytes) that this component's processes can use on the core device.

  This feature is available for v2.4.0 and later of the Greengrass nucleus component. AWS IoT Greengrass doesn't currently support this feature on Windows core devices.

Example basic configuration update

The following example `components` object specifies to deploy a component, `com.example.PythonRuntime`, that expects a configuration parameter named `pythonVersion`.

```
{
  "targetArn": "targetArn",
  "deploymentName": "deploymentName",
  "components": {
    "com.example.PythonRuntime": {
      "componentVersion": "1.0.0",
      "configurationUpdate": {
        "merge": "{\"pythonVersion\":\"3.7\"}"
      }
    }
```

```
  }
}
```

Example configuration update with reset and merge updates

Consider an example industrial dashboard component, `com.example.IndustrialDashboard`, that has the following default configuration.

```
{
  "name": null,
  "mode": "REQUEST",
  "network": {
    "useHttps": true,
    "port": {
      "http": 80,
      "https": 443
    },
  },
  "tags": []
}
```

The following configuration update specifies the following instructions:

1. Reset the HTTPS setting to its default value (`true`).
2. Reset the list of industrial tags to an empty list.
3. Merge a list of industrial tags that identify temperature and pressure data streams for two boilers.

```
{
  "reset": [
    "/network/useHttps",
    "/tags"
  ],
  "merge": {
    "tags": [
      "/boiler/1/temperature",
      "/boiler/1/pressure",
      "/boiler/2/temperature",
      "/boiler/2/pressure"
    ]
```

```
  }
}
```

The following example `components` object specifies to deploy this industrial dashboard component and configuration update.

```
{
  "targetArn": "targetArn",
  "deploymentName": "deploymentName",
  "components": {
    "com.example.IndustrialDashboard": {
      "componentVersion": "1.0.0",
      "configurationUpdate": {
        "reset": [
          "/network/useHttps",
          "/tags"
        ],
        "merge": "{\"tags\":[\"/boiler/1/temperature\",\"/boiler/1/pressure\",\"/
boiler/2/temperature\",\"/boiler/2/pressure\"]}"
      }
    }
  }
}
```

5. (Optional) Define deployment policies for the deployment. You can configure when core devices can safely apply a deployment or what to do if a core device fails to apply the deployment. To do so, add a `deploymentPolicies` object to `deployment.json`, and then do any of the following:

   1. (Optional) Specify the component update policy (`componentUpdatePolicy`). This policy defines whether or not the deployment lets components defer an update until they are ready to update. For example, components may need to clean up resources or finish critical actions before they can restart to apply an update. This policy also defines the amount of time that components have to respond to an update notification.

      This policy is an object with the following parameters:

      - `action` – (Optional) Whether or not to notify components and wait for them to report when they're ready to update. Choose from the following options:

- `NOTIFY_COMPONENTS` – The deployment notifies each component before it stops and updates that component. Components can use the SubscribeToComponentUpdates IPC operation to receive these notifications.
- `SKIP_NOTIFY_COMPONENTS` – The deployment doesn't notify components or wait for them to be safe to update.

  Defaults to `NOTIFY_COMPONENTS`.

- `timeoutInSeconds` The amount of time in seconds that each component has to respond to an update notification with the DeferComponentUpdate IPC operation. If the component doesn't respond within this amount of time, then the deployment proceeds on the core device.

  Defaults to 60 seconds.

2. (Optional) Specify the configuration validation policy (`configurationValidationPolicy`). This policy defines how long each component has to validate a configuration update from a deployment. Components can use the SubscribeToValidateConfigurationUpdates IPC operation to subscribe to notifications for their own configuration updates. Then, components can use the SendConfigurationValidityReport IPC operation to tell the AWS IoT Greengrass Core software if the configuration update is valid. If the configuration update isn't valid, the deployment fails.

   This policy is an object with the following parameter:

   - `timeoutInSeconds` (Optional) The amount of time in seconds that each component has to validate a configuration update. If the component doesn't respond within this amount of time, then the deployment proceeds on the core device.

     Defaults to 30 seconds.

3. (Optional) Specify the failure handling policy (`failureHandlingPolicy`). This policy is a string that defines whether or not to roll back devices if the deployment fails. Choose from the following options:

   - `ROLLBACK` – If the deployment fails on a core device, then the AWS IoT Greengrass Core software rolls back that core device to its previous configuration.
   - `DO_NOTHING` – If the deployment fails on a core device, then the AWS IoT Greengrass Core software keeps the new configuration. This can result in broken components if the new configuration isn't valid.

Defaults to ROLLBACK.

> **Important**
>
> The AWS IoT Greengrass Core software ignores the failure handling policy for bootstrap deployments. A bootstrap deployment updates the Greengrass nucleus version or includes a component with a bootstrap lifecycle step. If a bootstrap deployment fails, the AWS IoT Greengrass Core software rolls back the core device to its previous configuration. The rollback happens even if you set the failure handling policy to DO_NOTHING. This behavior keeps the core device in a working state after a restart.

Your deployment in `deployment.json` may look similar to the following example:

```
{
  "targetArn": "targetArn",
  "deploymentName": "deploymentName",
  "components": {
    "com.example.IndustrialDashboard": {
      "componentVersion": "1.0.0",
      "configurationUpdate": {
        "reset": [
          "/network/useHttps",
          "/tags"
        ],
        "merge": "{\"tags\":[\"/boiler/1/temperature\",\"/boiler/1/pressure\",\"/
boiler/2/temperature\",\"/boiler/2/pressure\"]}"
      }
    }
  },
  "deploymentPolicies": {
    "componentUpdatePolicy": {
      "action": "NOTIFY_COMPONENTS",
      "timeoutInSeconds": 30
    },
    "configurationValidationPolicy": {
      "timeoutInSeconds": 60
    },
    "failureHandlingPolicy": "ROLLBACK"
```

```
  }
}
```

6. (Optional) Define how the deployment stops, rolls out, or times out. AWS IoT Greengrass uses AWS IoT Core jobs to send deployments to core devices, so these options are identical to the configuration options for AWS IoT Core jobs. For more information, see Job rollout and abort configuration in the *AWS IoT Developer Guide*.

   To define the job options, add an `iotJobConfiguration` object to `deployment.json`. Then, define the options to configure.

   Your deployment in `deployment.json` may look similar to the following example:

```
{
  "targetArn": "targetArn",
  "deploymentName": "deploymentName",
  "components": {
    "com.example.IndustrialDashboard": {
      "componentVersion": "1.0.0",
      "configurationUpdate": {
        "reset": [
          "/network/useHttps",
          "/tags"
        ],
        "merge": "{\"tags\":[\"/boiler/1/temperature\",\"/boiler/1/pressure\",\"/
boiler/2/temperature\",\"/boiler/2/pressure\"]}"
      }
    }
  },
  "deploymentPolicies": {
    "componentUpdatePolicy": {
      "action": "NOTIFY_COMPONENTS",
      "timeoutInSeconds": 30
    },
    "configurationValidationPolicy": {
      "timeoutInSeconds": 60
    },
    "failureHandlingPolicy": "ROLLBACK"
  },
  "iotJobConfiguration": {
    "abortConfig": {
      "criteriaList": [
        {
```

```
          "action": "CANCEL",
          "failureType": "ALL",
          "minNumberOfExecutedThings": 100,
          "thresholdPercentage": 5
        }
      ]
    },
    "jobExecutionsRolloutConfig": {
      "exponentialRate": {
        "baseRatePerMinute": 5,
        "incrementFactor": 2,
        "rateIncreaseCriteria": {
          "numberOfNotifiedThings": 10,
          "numberOfSucceededThings": 5
        }
      },
      "maximumPerMinute": 50
    },
    "timeoutConfig":  {
      "inProgressTimeoutInMinutes": 5
    }
  }
}
```

7. (Optional) Add tags (`tags`) for the deployment. For more information, see Tag your AWS IoT Greengrass Version 2 resources.
8. Run the following command to create the deployment from `deployment.json`.

   ```
   aws greengrassv2 create-deployment --cli-input-json file://deployment.json
   ```

   The response includes a `deploymentId` that identifies this deployment. You can use the deployment ID to check the status of the deployment. For more information, see Check deployment status.

Update component configurations

Component configurations are JSON objects that define the parameters for each component. Each component's recipe defines its default configuration, which you modify when you deploy components to core devices.

When you create a deployment, you can specify the *configuration update* to apply for each component. Configuration updates are patch operations, which means that the update modifies the component configuration that exists on the core device. If the core device doesn't have the component, then the configuration update modifies and applies the default configuration for that deployment.

The configuration update defines *reset* updates and *merge* updates. Reset updates define which configuration values to reset to their defaults or remove. Merge updates define the new configuration values to set for the component. When you deploy a configuration update, the AWS IoT Greengrass Core software runs the reset update before the merge update.

Components can validate the configuration updates that you deploy. The component subscribes to receive a notification when a deployment changes its configuration, and it can reject a configuration that it doesn't support. For more information, see Interact with component configuration.

Topics

- Reset updates
- Merge updates
- Examples

Reset updates

Reset updates define which configuration values to reset to their default values on the core device. If a configuration value doesn't have a default value, then the reset update removes that value from the component's configuration. This can help you fix a component that breaks as the result of an invalid configuration.

Use a list of JSON pointers to define which configuration values to reset. JSON pointers start with a forward slash /. To identify a value in a nested component configuration, use forward slashes (/) to separate the keys for each level in the configuration. For more information, see the JSON pointer specification.

> **Note**
>
> You can reset only an entire list to its default values. You can't use reset updates to reset an individual element in a list.

To reset a component's entire configuration to its default values, specify a single empty string as the reset update.

```
"reset": [""]
```

Merge updates

Merge updates define the configuration values to insert into the component configuration on the core. The merge update is a JSON object that the AWS IoT Greengrass Core software merges after it resets the values in the paths that you specify in the reset update. When you use the AWS CLI or AWS SDKs, you must serialize this JSON object as a string.

You can merge a key-value pair that doesn't exist in the component's default configuration. You can also merge a key-value pair that has a different type than the value with the same key. The new value replaces the old value. This means that you can change the configuration object's structure.

You can merge null values and empty strings, lists, and objects.

> **Note**
>
> You can't use merge updates for the purpose of inserting or appending an element to a list. You can replace an entire list, or you can define an object where each element has a unique key.
> AWS IoT Greengrass uses JSON for configuration values. JSON specifies a number type but doesn't differentiate between integers and floats. As a result, configuration values might convert to floats in AWS IoT Greengrass. To ensure that your component uses the correct data type, we recommend that you define numeric configuration values as strings. Then, have your component parse them as integers or floats. This ensures that your configuration values have the same type in the configuration and on your core device.

Use recipe variables in merge updates

This feature is available for v2.6.0 and later of the Greengrass nucleus component.

If you set the Greengrass nucleus' interpolateComponentConfiguration configuration option to `true`, you can use recipe variables, other than the *`component_dependency_name`*`:configuration:`*`json_pointer`* recipe variable, in merge

updates. For example, you can use the `{iot:thingName}` recipe variable in a merge update to include the core device's AWS IoT thing name in a component configuration value, such as an interprocess communication (IPC) authorization policy.

Examples

The following example demonstrates configuration updates for a dashboard component that has the following default configuration. This example component displays information about industrial equipment.

```
{
  "name": null,
  "mode": "REQUEST",
  "network": {
    "useHttps": true,
    "port": {
      "http": 80,
      "https": 443
    },
  },
  "tags": []
}
```

Industrial dashboard component recipe

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.IndustrialDashboard",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "Displays information about industrial equipment.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "name": null,
      "mode": "REQUEST",
      "network": {
        "useHttps": true,
        "port": {
          "http": 80,
          "https": 443
```

```
        },
      },
      "tags": []
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "Run": "python3 -u {artifacts:path}/industrial_dashboard.py"
      }
    },
    {
      "Platform": {
        "os": "windows"
      },
      "Lifecycle": {
        "Run": "py -3 -u {artifacts:path}/industrial_dashboard.py"
      }
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.IndustrialDashboard
ComponentVersion: '1.0.0'
ComponentDescription: Displays information about industrial equipment.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    name: null
    mode: REQUEST
    network:
      useHttps: true
      port:
        http: 80
        https: 443
    tags: []
```

```
Manifests:
  - Platform:
      os: linux
    Lifecycle:
      Run: |
        python3 -u {artifacts:path}/industrial_dashboard.py
  - Platform:
      os: windows
    Lifecycle:
      Run: |
        py -3 -u {artifacts:path}/industrial_dashboard.py
```

Example 1: Merge update

You create a deployment that applies the following configuration update, which specifies a merge update but not a reset update. This configuration update tells the component to display the dashboard on HTTP port 8080 with data from two boilers.

Console

Configuration to merge

```
{
  "name": "Factory 2A",
  "network": {
    "useHttps": false,
    "port": {
      "http": 8080
    }
  },
  "tags": [
    "/boiler/1/temperature",
    "/boiler/1/pressure",
    "/boiler/2/temperature",
    "/boiler/2/pressure"
  ]
}
```

AWS CLI

The following command creates a deployment to a core device.

```
aws greengrassv2 create-deployment --cli-input-json file://dashboard-deployment.json
```

The `dashboard-deployment.json` file contains the following JSON document.

```
{
  "targetArn": "arn:aws:iot:us-west-2:123456789012:thing/MyGreengrassCore",
  "deploymentName": "Deployment for MyGreengrassCore",
  "components": {
    "com.example.IndustrialDashboard": {
      "componentVersion": "1.0.0",
      "configurationUpdate": {
        "merge": "{\"name\":\"Factory 2A\",\"network\":{\"useHttps\":false,\"port
\":{\"http\":8080}},\"tags\":[\"/boiler/1/temperature\",\"/boiler/1/pressure\",\"/
boiler/2/temperature\",\"/boiler/2/pressure\"]}"
      }
    }
  }
}
```

Greengrass CLI

The following Greengrass CLI command creates a local deployment on a core device.

```
sudo greengrass-cli deployment create \
  --recipeDir recipes \
  --artifactDir artifacts \
  --merge "com.example.IndustrialDashboard=1.0.0" \
  --update-config dashboard-configuration.json
```

The `dashboard-configuration.json` file contains the following JSON document.

```
{
  "com.example.IndustrialDashboard": {
    "MERGE": {
      "name": "Factory 2A",
      "network": {
        "useHttps": false,
        "port": {
          "http": 8080
        }
      },
      "tags": [
```

```
        "/boiler/1/temperature",
        "/boiler/1/pressure",
        "/boiler/2/temperature",
        "/boiler/2/pressure"
      ]
    }
  }
}
```

After this update, the dashboard component has the following configuration.

```
{
  "name": "Factory 2A",
  "mode": "REQUEST",
  "network": {
    "useHttps": false,
    "port": {
      "http": 8080,
      "https": 443
    }
  },
  "tags": [
    "/boiler/1/temperature",
    "/boiler/1/pressure",
    "/boiler/2/temperature",
    "/boiler/2/pressure"
  ]
}
```

Example 2: Reset and merge updates

Then, you create a deployment that applies the following configuration update, which specifies a reset update and a merge update. These updates specify to display the dashboard on the default HTTPS port with data from different boilers. These updates modify the configuration that results from the configuration updates in the previous example.

Console

Reset paths

```
[
```

```
  "/network/useHttps",
  "/tags"
]
```

Configuration to merge

```
{
  "tags": [
    "/boiler/3/temperature",
    "/boiler/3/pressure",
    "/boiler/4/temperature",
    "/boiler/4/pressure"
  ]
}
```

AWS CLI

The following command creates a deployment to a core device.

```
aws greengrassv2 create-deployment --cli-input-json file://dashboard-
deployment2.json
```

The `dashboard-deployment2.json` file contains the following JSON document.

```
{
  "targetArn": "arn:aws:iot:us-west-2:123456789012:thing/MyGreengrassCore",
  "deploymentName": "Deployment for MyGreengrassCore",
  "components": {
    "com.example.IndustrialDashboard": {
      "componentVersion": "1.0.0",
      "configurationUpdate": {
        "reset": [
          "/network/useHttps",
          "/tags"
        ],
        "merge": "{\"tags\":[\"/boiler/3/temperature\",\"/boiler/3/pressure\",\"/
boiler/4/temperature\",\"/boiler/4/pressure\"]}"
      }
    }
  }
}
```

Greengrass CLI

The following Greengrass CLI command creates a local deployment on a core device.

```
sudo greengrass-cli deployment create \
  --recipeDir recipes \
  --artifactDir artifacts \
  --merge "com.example.IndustrialDashboard=1.0.0" \
  --update-config dashboard-configuration2.json
```

The `dashboard-configuration2.json` file contains the following JSON document.

```
{
  "com.example.IndustrialDashboard": {
    "RESET": [
      "/network/useHttps",
      "/tags"
    ],
    "MERGE": {
      "tags": [
        "/boiler/3/temperature",
        "/boiler/3/pressure",
        "/boiler/4/temperature",
        "/boiler/4/pressure"
      ]
    }
  }
}
```

After this update, the dashboard component has the following configuration.

```
{
  "name": "Factory 2A",
  "mode": "REQUEST",
  "network": {
    "useHttps": true,
    "port": {
      "http": 8080,
      "https": 443
    }
  },
```

```
  "tags": [
    "/boiler/3/temperature",
    "/boiler/3/pressure",
    "/boiler/4/temperature",
    "/boiler/4/pressure",
  ]
}
```

Create subdeployments

> **Note**
>
> The subdeployment feature is available on Greengrass nucleus version 2.9.0 and later. It is not possible to deploy a configuration to a subdeployment with earlier component versions of Greengrass nucleus.

A subdeployment is a deployment that targets a smaller subset of devices within a parent deployment. You can use subdeployments to deploy a configuration to a smaller subset of devices. You can also create subdeployments to retry an unsuccessful parent deployment when one or more devices in that parent deployment fails. With this feature, you can select devices that failed in that parent deployment and create a subdeployment to test configurations until the subdeployment is successful. Once the subdeployment is successful, you can redeploy that configuration to the parent deployment.

Follow the steps in this section to create a subdeployment and check its status. For more information about how to create deployments, see Create deployments.

To create a subdeployment (AWS CLI)

1. Run the following command to retrieve the latest deployments for a thing group. Replace the ARN in the command with the ARN of the thing group to query. Set `--history-filter` to **`LATEST_ONLY`** to see the latest deployment of that thing group.

   ```
   aws greengrassv2 list-deployments --target-arn arn:aws:iot:region:account-id:thinggroup/thingGroupName --history-filter LATEST_ONLY
   ```

2. Copy the `deploymentId` from the response to the **list-deployments** command to use in the next step.

3. Run the following command to retrieve the status of a deployment. Replace *deploymentId* with the ID of the deployment to query.

```
aws greengrassv2 get-deployment --deployment-id deploymentId
```

4. Copy the `iotJobId` from the response to the **get-deployment** command to use in the following step.

5. Run the following command to retrieve the list of job executions for the specified job. Replace *jobID* with the `iotJobId` from the previous step. Replace *status* with the status you want to filter for. You can filter results with the following statuses:

   - `QUEUED`
   - `IN_PROGRESS`
   - `SUCCEEDED`
   - `FAILED`
   - `TIMED_OUT`
   - `REJECTED`
   - `REMOVED`
   - `CANCELED`

```
aws iot list-job-executions-for-job --job-id jobID --status status
```

6. Create a new AWS IoT thing group, or use an existing thing group, for your subdeployment. Then, add an AWS IoT thing to this thing group. You use thing groups to manage fleets of Greengrass core devices. When you deploy software components to your devices, you can target either individual devices or groups of devices. You can add a device to a thing group with an active Greengrass deployment. Once added, you can then deploy that thing group's software components to that device.

   To create a new thing group and add your devices to it, do the following:

   a. Create an AWS IoT thing group. Replace *MyGreengrassCoreGroup* with the name for the new thing group. You can't use a colon (:) in a thing group name.

> **Note**
>
> If a thing group for a subdeployment is used with one `parentTargetArn`, it can't be reused with a different parent fleet. If a thing group has already been used to create a subdeployment for another fleet, the API will return an error.

```
aws iot create-thing-group --thing-group-name MyGreengrassCoreGroup
```

If the request succeeds, the response looks similar to the following example:

```
{
  "thingGroupName": "MyGreengrassCoreGroup",
  "thingGroupArn": "arn:aws:iot:us-
west-2:123456789012:thinggroup/MyGreengrassCoreGroup",
  "thingGroupId": "4df721e1-ff9f-4f97-92dd-02db4e3f03aa"
}
```

b. Add a provisioned Greengrass core to your thing group. Run the following command with these parameters:

- Replace *MyGreengrassCore* with the name of your provisioned Greengrass core.
- Replace *MyGreengrassCoreGroup* with the name of your thing group.

```
aws iot add-thing-to-thing-group --thing-name MyGreengrassCore --thing-group-
name MyGreengrassCoreGroup
```

The command doesn't have any output if the request succeeds.

7. Create a file called `deployment.json`, and then copy the following JSON object into the file. Replace *targetArn* with the ARN of the AWS IoT thing group to target for the subdeployment. A subdeployment target can only be a thing group. Thing group ARNs have the following format:

- **Thing group** – `arn:aws:iot:`*region*`:`*account-id*`:thinggroup/`*thingGroupName*

```
{
```

```
  "targetArn": "targetArn"
}
```

8. Run the following command again to get the original deployment's details. These details include metadata, components, and job configuration. Replace *deploymentId* with the ID from Step 1. You can use this deployment configuration to configure your subdeployment and make changes as needed.

   ```
   aws greengrassv2 get-deployment --deployment-id deploymentId
   ```

   The response contains the deployment's details. Copy any of the following key-value pairs from the **get-deployment** command's response into `deployment.json`. You can change these values for the subdeployment. For more information about the details of this command, see GetDeployment.

   - `components` – The deployment's components. To uninstall a component, remove it from this object. If the component defines an uninstall lifecycle step, the AWS IoT Greengrass Core software runs the uninstall script before removing the component from the device.
   - `deploymentName` – The deployment's name.
   - `deploymentPolicies` – The deployment's policies.
   - `iotJobConfiguration` – The deployment's job configuration.
   - `parentTargetArn` – The target of the parent deployment.
   - `tags` – The deployment's tags.

9. Run the following command to create the subdeployment from `deployment.json`. Replace *subdeploymentName* with a name for the subdeployment.

   ```
   aws greengrassv2 create-deployment --deployment-name subdeploymentName --cli-input-
   json file://deployment.json
   ```

   The response includes a `deploymentId` that identifies this subdeployment. You can use the deployment ID to check the status of the deployment. For more information, see Check deployment status.

10. If the subdeployment is successful, you can use its configuration to revise the parent deployent. Copy the `deployment.json` that you used in the previous step. Replace the `targetArn` in the JSON file with the parent deployment's ARN and run the following command to create the parent deployment using this new configuration.

> **Note**
>
> If you create a new deployment revision of the parent fleet, it replaces all deployment revisions and subdeployments for that parent deployment. For more information, see Revise deployments.

```
aws greengrassv2 create-deployment --cli-input-json file://deployment.json
```

The response includes a `deploymentId` that identifies this deployment. You can use the deployment ID to check the status of the deployment. For more information, see Check deployment status.

Revise deployments

Each target thing or thing group can have one active deployment at a time. When you create a deployment for a target that already has a deployment, the software components in the new deployment replace those from the previous deployment. If the new deployment doesn't define a component that the previous deployment defines, the AWS IoT Greengrass Core software removes that component from the target core devices. You can revise an existing deployment so that you don't remove the components that run on core devices from a previous deployment to a target.

To revise a deployment, you create a deployment that starts from the same components and configurations that exist in a previous deployment. You use the CreateDeployment operation, which is the same operation that you use to create deployments.

To revise a deployment (AWS CLI)

1. Run the following command to list the deployments for the deployment target. Replace *`targetArn`* with the ARN of the target AWS IoT thing or thing group.

   ```
   aws greengrassv2 list-deployments --target-arn targetArn
   ```

   The response contains a list with the latest deployment for the target. Copy the `deploymentId` from the response to use in the next step.

> **Note**
>
> You can also revise a deployment other than the latest revision for the target. Specify the `--history-filter ALL` argument to list all deployments for the target. Then, copy the ID of the deployment that you want to revise.

2. Run the following command to get the deployment's details. These details include metadata, components, and job configuration. Replace *deploymentId* with the ID from the previous step.

   ```
   aws greengrassv2 get-deployment --deployment-id deploymentId
   ```

   The response contains the deployment's details.

3. Create a file called `deployment.json` and copy the previous command's response into the file.

4. Remove the following key-value pairs from the JSON object in `deployment.json`:

   - `deploymentId`
   - `revisionId`
   - `iotJobId`
   - `iotJobArn`
   - `creationTimestamp`
   - `isLatestForTarget`
   - `deploymentStatus`

   The CreateDeployment operation expects a payload with the following structure.

   ```
   {
     "targetArn": "String",
     "components": Map of components,
     "deploymentPolicies": DeploymentPolicies,
     "iotJobConfiguration": DeploymentIoTJobConfiguration,
     "tags": Map of tags
   }
   ```

5. In `deployment.json`, do any of the following:

- Change the deployment's name (`deploymentName`).
- Change the deployment's components (`components`).
- Change the deployment's policies (`deploymentPolicies`).
- Change the deployment's job configuration (`iotJobConfiguration`).
- Change the deployment's tags (`tags`).

For more information about how to define these deployment details, see Create deployments.

6. Run the following command to create the deployment from `deployment.json`.

```
aws greengrassv2 create-deployment --cli-input-json file://deployment.json
```

The response includes a `deploymentId` that identifies this deployment. You can use the deployment ID to check the status of the deployment. For more information, see Check deployment status.

Cancel deployments

You can cancel an active deployment to prevent its software components from installing on AWS IoT Greengrass core devices. If you cancel a deployment that targets a thing group, core devices that you add to the group won't receive that continuous deployment. If a core device already runs the deployment, you won't change the components on that device when you cancel the deployment. You must create a new deployment or revise the deployment to modify the components that run on the core devices that received the canceled deployment.

To cancel a deployment (AWS CLI)

1. Run the following command to find the ID of the latest deployment revision for a target. The latest revision is the only deployment that can be active for a target, because previous deployments cancel when you create a new revision. Replace *`targetArn`* with the ARN of the target AWS IoT thing or thing group.

```
aws greengrassv2 list-deployments --target-arn targetArn
```

The response contains a list with the latest deployment for the target. Copy the `deploymentId` from the response to use in the next step.

2. Run the following command to cancel the deployment. Replace *deploymentId* with the ID from the previous step.

   ```
   aws greengrassv2 cancel-deployment --deployment-id deploymentId
   ```

   If the operation succeeds, the deployment status changes to CANCELED.

Check deployment status

You can check the status of a deployment that you create in AWS IoT Greengrass. You can also check the status of the AWS IoT jobs that roll out the deployment to each core device. While a deployment is active, the AWS IoT job's status is IN_PROGRESS. After you create a new revision of a deployment, the status of the previous revision's AWS IoT job changes to CANCELLED.

Topics

- Check deployment status
- Check device deployment status

Check deployment status

You can check the status of a deployment that you identify by its target or its ID.

To check deployment status by target (AWS CLI)

- Run the following command to retrieve the status of the latest deployment for a target. Replace *targetArn* with the Amazon Resource Name (ARN) of the AWS IoT thing or thing group that the deployment targets.

  ```
  aws greengrassv2 list-deployments --target-arn targetArn
  ```

  The response contains a list with the latest deployment for the target. This deployment object includes the status of the deployment.

To check deployment status by ID (AWS CLI)

- Run the following command to retrieve the status of a deployment. Replace *deploymentId* with the ID of the deployment to query.

```
aws greengrassv2 get-deployment --deployment-id deploymentId
```

The response contains the status of the deployment.

Check device deployment status

You can check the status of a deployment job that applies to an individual core device. You can also check the status of a deployment job for a thing group deployment.

To check deployment job statuses for a core device (AWS CLI)

- Run the following command to retrieve the status of all deployment jobs for a core device. Replace *coreDeviceName* with the name of the core device to query.

  ```
  aws greengrassv2 list-effective-deployments --core-device-thing-name coreDeviceName
  ```

  The response contains the list of deployment jobs for the core device. You can identify the job for a deployment by the job's `deploymentId` or `targetArn`. Each deployment job contains the status of the job on the core device.

To check deployment statuses for a thing group (AWS CLI)

1. Run the following command to retrieve the ID of an existing deployment. Replace *targetArn* with the ARN of the target thing group.

   ```
   aws greengrassv2 list-deployments --target-arn targetArn
   ```

   The response contains a list with the latest deployment for the target. Copy the `deploymentId` from the response to use in the next step.

   > **Note**
   >
   > You can also list a deployment other than the latest deployment for the target. Specify the `--history-filter ALL` argument to list all deployments for the target. Then, copy the ID of the deployment that you want to check the status of.

2. Run the following command to get the deployment's details. Replace *deploymentID* with the ID from the previous step.

```
aws greengrassv2 get-deployment --deployment-id deploymentId
```

   The response contains information about the deployment. Copy the `iotJobId` from the response to use in the following step.

3. Run the following command to describe a core device's job execution for the deployment. Replace *iotJobId* and *coreDeviceThingName* with the job ID from the previous step and the core device you want to check the status for.

```
aws iot describe-job-execution --job-id iotJobId --thing-name coreDeviceThingName
```

   The response contains the status of the core device's deployment job execution and details about the status. The `detailsMap` contains the following information:

   - `detailed-deployment-status` – The deployment result status, which can be one of the following values:
     - `SUCCESSFUL` – The deployment succeeded.
     - `FAILED_NO_STATE_CHANGE` – The deployment failed while the core device prepared to apply the deployment.
     - `FAILED_ROLLBACK_NOT_REQUESTED` – The deployment failed, and the deployment didn't specify to roll back to a previous working configuration, so the core device might not be functioning correctly.
     - `FAILED_ROLLBACK_COMPLETE` – The deployment failed, and the core device successfully rolled back to a previous working configuration.
     - `FAILED_UNABLE_TO_ROLLBACK` – The deployment failed, and the core device failed to roll back to a previous working configuration, so the core device might not be functioning correctly.

   > **Note**
   >
   > The AWS IoT Greengrass Core software ignores the failure handling policy for bootstrap deployments. A bootstrap deployment updates the Greengrass nucleus version or includes a component with a bootstrap lifecycle step. These

deployments always roll back on failure and report `FAILED_ROLLBACK_COMPLETE` or `FAILED_UNABLE_TO_ROLLBACK`. For more information, see Create deployments.

If the deployment failed, check the `deployment-failure-cause` value and the core device's log files to identify the issue. For more information about how to access the core device's log files, see Monitor AWS IoT Greengrass logs.

- `deployment-failure-cause` – An error message that provides additional details about why the job execution failed.

The response looks similar to the following example.

```
{
  "execution": {
    "jobId": "2cc2698a-5175-48bb-adf2-1dd345606ebd",
    "status": "FAILED",
    "statusDetails": {
      "detailsMap": {
        "deployment-failure-cause": "No local or cloud component version satisfies
 the requirements. Check whether the version constraints conflict and that
 the component exists in your AWS account with a version that matches the
 version constraints. If the version constraints conflict, revise deployments
 to resolve the conflict. Component com.example.HelloWorld version constraints:
 LOCAL_DEPLOYMENT requires =1.0.0, thinggroup/MyGreengrassCoreGroup requires
 =1.0.1.",
        "detailed-deployment-status": "FAILED_NO_STATE_CHANGE"
      }
    },
    "thingArn": "arn:aws:iot:us-west-2:123456789012:thing/MyGreengrassCore",
    "queuedAt": "2022-02-15T14:45:53.098000-08:00",
    "startedAt": "2022-02-15T14:46:05.670000-08:00",
    "lastUpdatedAt": "2022-02-15T14:46:20.892000-08:00",
    "executionNumber": 1,
    "versionNumber": 3
  }
}
```

Logging and monitoring in AWS IoT Greengrass

Monitoring is an important part of maintaining the reliability, availability, and performance of AWS IoT Greengrass and your AWS solutions. You should collect monitoring data from all parts of your AWS solution so that you can more easily debug a multi-point failure, if one occurs. Before you start monitoring AWS IoT Greengrass, you should create a monitoring plan that includes answers to the following questions:

- What are your monitoring goals?
- Which resources will you monitor?
- How often will you monitor these resources?
- Which monitoring tools will you use?
- Who will perform the monitoring tasks?
- Who should be notified when something goes wrong?

Topics

- Monitoring tools
- Monitor AWS IoT Greengrass logs
- Log AWS IoT Greengrass V2 API calls with AWS CloudTrail
- Gather system health telemetry data from AWS IoT Greengrass core devices
- Get deployment and component health status notifications
- Check Greengrass core device status

Monitoring tools

AWS provides tools that you can use to monitor AWS IoT Greengrass. You can configure some of these tools to do the monitoring for you. Some of the tools require manual intervention. We recommend that you automate monitoring tasks as much as possible.

You can use the following automated monitoring tools to monitor AWS IoT Greengrass and report issues:

- **Amazon CloudWatch Logs** – Monitor, store, and access your log files from AWS CloudTrail or other sources. For more information, see Monitoring log files in the *Amazon CloudWatch User Guide*.
- **AWS CloudTrail Log Monitoring** – Share log files between accounts, monitor CloudTrail log files in real time by sending them to CloudWatch Logs, write log processing applications in Java, and validate that your log files have not changed after delivery by CloudTrail. For more information, see Working with CloudTrail log files in the *AWS CloudTrail User Guide*.
- **Greengrass system health telemetry** – Subscribe to receive telemetry data sent from the Greengrass core. For more information, see the section called "Gather system health telemetry data".
- **Device health notifications** Create events using Amazon EventBridge to receive status updates regarding deployments and components. For more information, see Get deployment and component health status notifications.
- **Fleet status service** – Use the fleet status API operations to check the status of core devices and their Greengrass components. You can also view fleet status information in the AWS IoT Greengrass console. For more information, see Check Greengrass core device status.

Monitor AWS IoT Greengrass logs

AWS IoT Greengrass consists of the cloud service and the AWS IoT Greengrass Core software. The AWS IoT Greengrass Core software can write logs to Amazon CloudWatch Logs and to the core device's local file system. Greengrass components that run on the core device can also write logs to CloudWatch Logs and the local file system. You can use logs to monitor events and troubleshoot issues. All AWS IoT Greengrass log entries include a timestamp, log level, and information about the event.

By default, the AWS IoT Greengrass Core software writes logs to only the local file system. You can view file system logs in real time, so you can debug Greengrass components that you develop and deploy. You can also configure a core device to write logs to CloudWatch Logs, so you can troubleshoot the core device without access to the local file system. For more information, see Enable logging to CloudWatch Logs.

Topics

- Access file system logs
- Access CloudWatch Logs
- Access system service logs

- Enable logging to CloudWatch Logs
- Configure logging for AWS IoT Greengrass
- AWS CloudTrail logs

Access file system logs

The AWS IoT Greengrass Core software stores logs in the */greengrass/v2*/logs folder on a core device, where */greengrass/v2* is the path to the AWS IoT Greengrass root folder. The logs folder has the following structure.

```
/greengrass/v2
### logs
    ### greengrass.log
    ### greengrass_2021_09_14_15_0.log
    ### ComponentName.log
    ### ComponentName_2021_09_14_15_0.log
    ### main.log
```

- `greengrass.log` – The AWS IoT Greengrass Core software log file. Use this log file to view real-time information about components and deployments. This log file includes logs for the Greengrass nucleus, which is the core of the AWS IoT Greengrass Core software, and plugin components, such as log manager and secret manager.
- *ComponentName*`.log` – Greengrass component log files. Use component log files to view real-time information about a Greengrass component that runs on the core device. Generic components and Lambda components write standard output (stdout) and standard error (stderr) to these log files.
- `main.log` – The log file for the `main` service that handles component lifecycles. This log file will always be empty.

For more information about the differences between plugin, generic, and Lambda components, see Component types.

The following considerations apply when you use file system logs:

- **Root user permissions**

  You must have root permissions to read AWS IoT Greengrass logs on the file system.

- **Log file rotation**

  The AWS IoT Greengrass Core software rotates log files every hour or when they exceed a file size limit. Rotated log files contain a timestamp in their file name. For example, a rotated AWS IoT Greengrass Core software log file might be named `greengrass_2021_09_14_15_0.log`. The default file size limit is 1,024 KB (1 MB). You can configure the file size limit on the Greengrass nucleus component.

- **Log file deletion**

  The AWS IoT Greengrass Core software cleans up earlier log files when the size of AWS IoT Greengrass Core software log files or Greengrass component log files, including rotated log files, exceeds a disk space limit. The default disk space limit for the AWS IoT Greengrass Core software log and each component log is 10,240 KB (10 MB). You can configure the AWS IoT Greengrass Core software log disk space limit on the Greengrass nucleus component or the log manager component. You can configure each component's log disk space limit on the log manager component.

To view the AWS IoT Greengrass Core software log file

- Run the following command to view the log file in real time. Replace */greengrass/v2* with the path to the AWS IoT Greengrass root folder.

  Linux or Unix

  ```
  sudo tail -f /greengrass/v2/logs/greengrass.log
  ```

  Windows Command Prompt (CMD)

  ```
  type C:\greengrass\v2\logs\com.example.HelloWorld.log
  ```

  The `type` command writes the file's contents to the terminal. Run this command multiple times to observe changes in the file.

  PowerShell

  ```
  gc C:\greengrass\v2\logs\greengrass.log -Tail 10 -Wait
  ```

To view the log file for a component

- Run the following command to view the log file in real time. Replace */greengrass/v2* or *C:\greengrass\v2* with the path to the AWS IoT Greengrass root folder, and replace *com.example.HelloWorld* with the name of the component.

  Linux or Unix

  ```
  sudo tail -f /greengrass/v2/logs/com.example.HelloWorld.log
  ```

  PowerShell

  ```
  gc C:\greengrass\v2\logs\com.example.HelloWorld.log -Tail 10 -Wait
  ```

You can also use the `logs` command of the Greengrass CLI to analyze Greengrass logs on a core device. To use the `logs` command, you must configure the Greengrass nucleus to output JSON format log files. For more information, see Greengrass Command Line Interface and logs.

Access CloudWatch Logs

You can deploy the log manager component to configure the core device to write to CloudWatch Logs. For more information, see Enable logging to CloudWatch Logs. Then, you can view logs on the **Logs** page of the Amazon CloudWatch console or using the CloudWatch Logs API.

Log group name

```
/aws/greengrass/componentType/region/componentName
```

The log group name uses the following variables:

- `componentType` – The type of the component, which can be one of the following:
  - `GreengrassSystemComponent` – This log group includes logs for the nucleus and plugin components, which run in the same JVM as the Greengrass nucleus. The component is part of the Greengrass nucleus.
  - `UserComponent` – This log group includes logs for generic components, Lambda components, and other applications on the device. The component isn't part of the Greengrass nucleus.

For more information, see Component types.

- `region` – The AWS Region that the core device uses.
- `componentName` – The name of the component. For system logs, this value is `System`.

Log stream name

```
/date/thing/thingName
```

The log stream name uses the following variables:

- `date` – The date of the log, such as `2020/12/15`. The log manager component uses the `yyyy/MM/dd` format.
- `thingName` – The name of the core device.

> **Note**
>
> If a thing name contains a colon (`:`), the log manager replaces the colon with a plus (+).

The following considerations apply when you use the log manager component to write to CloudWatch Logs:

- **Log delays**

> **Note**
>
> We recommend that you upgrade to log manager version 2.3.0 which reduces log delays for rotated and active log files. When you upgrade to log manager 2.3.0, we recommend you also upgrade to Greengrass nucleus 2.9.1.

The log manager component version 2.2.8 (and earlier) processes and uploads logs from only rotated log files. By default, the AWS IoT Greengrass Core software rotates log files every hour or after they are 1,024 KB. As a result, the log manager component uploads logs only after the AWS IoT Greengrass Core software or a Greengrass component writes over 1,024 KB worth of logs. You can configure a lower log file size limit to cause log files to rotate more often. This causes the log manager component to upload logs to CloudWatch Logs more frequently.

The log manager component version 2.3.0 (and later) processes and uploads all logs. When you write a new log, log manager version 2.3.0 (and later) processes and directly uploads that active log file instead of waiting for it to be rotated. This means that you can view the new log in 5 minutes or less.

The log manager component uploads new logs periodically. By default, the log manager component uploads new logs every 5 minutes. You can configure a lower upload interval, so the log manager component uploads logs to CloudWatch Logs more frequently by configuring the `periodicUploadIntervalSec`. For more information about how to configure this periodic interval, see Configuration.

Logs can be uploaded in near real-time from the same Greengrass file system. If you need to observe logs in real time, consider using file system logs.

> **Note**
>
> If you're using different file systems to write logs to, log manager reverts back to the behavior in log manager component versions 2.2.8 and earlier. For information about accessing file system logs, see Access file system logs.

- **Clock skew**

  The log manager component uses the standard Signature Version 4 signing process to create API requests to CloudWatch Logs. If the system time on a core device is out of sync by more than 15 minutes, then CloudWatch Logs rejects the requests. For more information, see Signature Version 4 signing process in the *AWS General Reference*.

Access system service logs

If you configure the AWS IoT Greengrass Core software as a system service, you can view system service logs to troubleshoot issues, such as the software failing to start.

To view system service logs (CLI)

1. Run the following command to view AWS IoT Greengrass Core software system service logs.

Linux or Unix (systemd)

```
sudo journalctl -u greengrass.service
```

Windows Command Prompt (CMD)

```
type C:\greengrass\v2\logs\greengrass.wrapper.log
```

PowerShell

```
gc C:\greengrass\v2\logs\greengrass.wrapper.log
```

2. On Windows devices, the AWS IoT Greengrass Core software creates a separate log file for system service errors. Run the following command to view the system service error logs.

   Windows Command Prompt (CMD)

   ```
   type C:\greengrass\v2\logs\greengrass.err.log
   ```

   PowerShell

   ```
   gc C:\greengrass\v2\logs\greengrass.err.log
   ```

On Windows devices, you can also use the **Event Viewer** application to view system service logs.

To view Windows service logs (Event Viewer)

1. Open the **Event Viewer** application.
2. Select **Windows Logs** to expand it.
3. Choose **Application** to view application service logs.
4. Find and open event logs whose **Source** is **greengrass**.

Enable logging to CloudWatch Logs

You can deploy the log manager component to configure a core device to write logs to CloudWatch Logs. You can enable CloudWatch Logs for AWS IoT Greengrass Core software logs, and you can enable CloudWatch Logs for specific Greengrass components.

Note

The Greengrass core device's token exchange role must allow the core device to write to CloudWatch Logs, as shown in the following example IAM policy. If you installed the AWS IoT Greengrass Core software with automatic resource provisioning, your core device has these permissions.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:DescribeLogStreams"
      ],
      "Effect": "Allow",
      "Resource": "arn:aws:logs:*:*:*"
    }
  ]
}
```

To configure a core device to write AWS IoT Greengrass Core software logs to CloudWatch Logs, create a deployment that specifies a configuration update that sets `uploadToCloudWatch` to `true` for the `aws.greengrass.LogManager` component. AWS IoT Greengrass Core software logs include logs for the Greengrass nucleus and plugin components.

```
{
  "logsUploaderConfiguration": {
    "systemLogsConfiguration": {
      "uploadToCloudWatch": "true"
    }
```

```
  }
}
```

To configure a core device to write a Greengrass component's logs to CloudWatch Logs, create a deployment that specifies a configuration update that adds the component to the list of component logging configurations. When you add a component to this list, the log manager component writes its logs to CloudWatch Logs. Component logs include logs for generic components and Lambda components.

```
{
  "logsUploaderConfiguration": {
    "componentLogsConfigurationMap": {
      "com.example.HelloWorld": {

      }
    }
  }
}
```

When you deploy the log manager component, you can also configure disk space limits and whether the core device deletes log files after writing them to CloudWatch Logs. For more information, see Configure logging for AWS IoT Greengrass.

Configure logging for AWS IoT Greengrass

You can configure the following options to customize logging for Greengrass core devices. To configure these options, create a deployment that specifies a configuration update to the Greengrass nucleus or log manager components.

- **Writing logs to CloudWatch Logs**

  To remotely troubleshoot core devices, you can configure core devices to write AWS IoT Greengrass Core software and component logs to CloudWatch Logs. To do so, deploy and configure the log manager component. For more information, see Enable logging to CloudWatch Logs.

- **Deleting uploaded log files**

  To reduce disk space usage, you can configure core devices to delete log files after writing the log files to CloudWatch Logs. For more information, see the log manager component's

`deleteLogFileAfterCloudUpload` parameter, which you can specify for AWS IoT Greengrass Core software logs and component logs.

- **Log disk space limits**

  To limit disk space usage, you can configure the maximum disk space for each log, including its rotated log files, on a core device. For example, you can configure the maximum combined disk space for `greengrass.log` and rotated `greengrass.log` files. For more information, see the Greengrass nucleus component's `logging.totalLogsSizeKB` parameter and the log manager component's `diskSpaceLimit` parameter, which you can specify for AWS IoT Greengrass Core software logs and component logs.

- **Log file size limit**

  You can configure the maximum file size for each log file. After a log file exceeds this file size limit, the AWS IoT Greengrass Core software creates a new log file. The log manager component version 2.28 (and earlier) writes only rotated log files to CloudWatch Logs, so you can specify a lower file size limit to write logs to CloudWatch Logs more frequently. The log manager component version 2.3.0 (and later) processes and uploads all logs instead of waiting for them to be rotated. For more information, see the Greengrass nucleus component's log file size limit parameter (`logging.fileSizeKB`).

- **Minimum log levels**

  You can configure the minimum log level that the Greengrass nucleus component writes to file system logs. For example, you might specify `DEBUG` level logs to help with troubleshooting, or you might specify `ERROR` level logs to reduce the amount of logs that a core device creates. For more information, see the Greengrass nucleus component's log level parameter (`logging.level`).

  You can also configure the minimum log level that the log manager component writes to CloudWatch Logs. For example, you might specify a higher log level to reduce logging costs. For more information, see the log manager component's `minimumLogLevel` parameter, which you can specify for AWS IoT Greengrass Core software logs and component logs.

- **Interval to check for logs to write to CloudWatch Logs**

  To increase or decrease how frequently the log manager component writes logs to CloudWatch Logs, you can configure the interval where it checks for new log files to write. For example, you might specify a lower interval to view logs in CloudWatch Logs sooner than you would with the default 5-minute interval. You might specify a higher interval to reduce costs, because the log

manager component batches log files into fewer requests. For more information, see the log manager component's upload interval parameter (`periodicUploadIntervalSec`).

- **Log format**

  You can choose whether the AWS IoT Greengrass Core software writes logs in text or JSON format. Choose text format if you read logs, or choose JSON format if you use an application to read or parse logs. For more information, see the Greengrass nucleus component's log format parameter (`logging.format`).

- **Local file system logs folder**

  You can change the logs folder from */greengrass/v2*`/logs` to another folder on the core device. For more information, see the Greengrass nucleus component's output directory parameter (`logging.outputDirectory`).

AWS CloudTrail logs

AWS IoT Greengrass integrates with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or AWS service in AWS IoT Greengrass. For more information, see Log AWS IoT Greengrass V2 API calls with AWS CloudTrail.

Log AWS IoT Greengrass V2 API calls with AWS CloudTrail

AWS IoT Greengrass V2 is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in AWS IoT Greengrass Version 2. CloudTrail captures all API calls for AWS IoT Greengrass as events. The calls that are captured include calls from the AWS IoT Greengrass console and code calls to the AWS IoT Greengrass API operations.

If you create a trail, you can enable continuous delivery of CloudTrail events to an S3 bucket, including events for AWS IoT Greengrass. If you don't configure a trail, you can still view the most recent events in the CloudTrail console in **Event history**. Using the information collected by CloudTrail, you can determine the request that was made to AWS IoT Greengrass, the IP address from which the request was made, who made the request, when it was made, and additional details.

For more information about CloudTrail, see the AWS CloudTrail User Guide.

Topics

- AWS IoT Greengrass V2 information in CloudTrail
- AWS IoT Greengrass data events in CloudTrail
- AWS IoT Greengrass management events in CloudTrail
- Understanding AWS IoT Greengrass V2 log file entries

AWS IoT Greengrass V2 information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When activity occurs in AWS IoT Greengrass, that activity is recorded in a CloudTrail event along with other AWS service events in **Event history**. You can view, search, and download recent events in your AWS account. For more information, see Viewing events with CloudTrail Event history.

For an ongoing record of events in your AWS account, including events for AWS IoT Greengrass, create a trail. A *trail* enables CloudTrail to deliver log files to an S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs. For more information, see the following:

- Overview for creating a trail
- CloudTrail supported services and integrations
- Configuring Amazon SNS notifications for CloudTrail
- Receiving CloudTrail log files from multiple regions and Receiving CloudTrail log files from multiple accounts

All AWS IoT Greengrass V2 actions are logged by CloudTrail and are documented in the AWS IoT Greengrass V2 API Reference. For example, calls to the `CreateComponentVersion`, `CreateDeployment` and `CancelDeployment` actions generate entries in the CloudTrail log files.

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or AWS Identity and Access Management (IAM) user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.

- Whether the request was made by another AWS service.

For more information, see the CloudTrail userIdentity element.

AWS IoT Greengrass data events in CloudTrail

Data events provide information about the resource operations performed on or in a resource (for example, getting a component version or the configuration of a deployment). These are also known as data plane operations. Data events are often high-volume activities. By default, CloudTrail doesn't log data events. The CloudTrail **Event history** doesn't record data events.

Additional charges apply for data events. For more information about CloudTrail pricing, see AWS CloudTrail Pricing.

You can log data events for the AWS IoT Greengrass resource types by using the CloudTrail console, AWS CLI, or CloudTrail API operations. The table in this section shows the resource types available for AWS IoT Greengrass.

- To log data events using the CloudTrail console, create a trail or event data store to log data events, or update an existing trail or event data store to log data events.
  1. Choose **Data events** to log data events.
  2. From the **Data event type** list, choose the resource type for which you want to log data events.
  3. Choose the log selector template you want to use. You can log all data events for the resource type, log all `readOnly` events, log all `writeOnly` events, or create a custom log selector template to filter on the `readOnly`, `eventName`, and `resources.ARN` fields.
- To log data events using the AWS CLI, configure the `--advanced-event-selectors` parameter to set the `eventCategory` field equal to `Data` and the `resources.type` field equal to the resource type value (see table). You can add conditions to filter on the values of the `readOnly`, `eventName`, and `resources.ARN` fields.
  - To configure a trail to log data events, run the put-event-selectors command. For more information, see Logging data events for trails with the AWS CLI.
  - To configure an event data store to log data events, run the create-event-data-store command to create a new event data store to log data events, or run the update-event-data-store command to update an existing event data store. For more information, see Logging data events for event data stores with the AWS CLI.

The following table lists the AWS IoT Greengrass resource types. The **Data event type (console)** column shows the value to choose from the **Data event type** list on the CloudTrail console. The **resources.type value** column shows the `resources.type` value, which you would specify when configuring advanced event selectors using the AWS CLI or CloudTrail APIs. The **Data APIs logged to CloudTrail** column shows the API calls logged to CloudTrail for the resource type.

Data event type (console)	resources.type value	Data APIs logged to CloudTrail
IoT certificate	`AWS::IoT::Certificate`	• VerifyClientDeviceIdentity • VerifyClientDeviceIoTCertificateAssociation
IoT Greengrass component version	`AWS::GreengrassV2::ComponentVersion`	• ResolveComponentCandidates
IoT Greengrass deployment	`AWS::GreengrassV2::Deployment`	• GetDeploymentConfiguration
IoT thing	`AWS::IoT::Thing`	• ListThingGroupsForCoreDevices • PutCertificateAuthorities • VerifyClientDeviceIoTCertificateAssociation

You can configure advanced event selectors to filter on the `eventName`, `readOnly`, and `resources.ARN` fields to log only those events that are important to you.

Add a filter on `eventName` to include or exclude specific data APIs.

For more information about these fields, see AdvancedFieldSelector.

The following examples show how to configure advanced selectors using the AWS CLI. Replace *`TrailName`* and *`region`* with your own information.

Example– Log data events for IoT things

```
aws cloudtrail put-event-selectors --trail-name TrailName --region region \
```

```
--advanced-event-selectors \
'[
    {
            "Name": "Log all thing data events",
            "FieldSelectors": [
                { "Field": "eventCategory", "Equals": ["Data"] },
                { "Field": "resources.type", "Equals": ["AWS::IoT::Thing"] }
            ]
        }
]'
```

Example– Filter on a specific IoT thing API

```
aws cloudtrail put-event-selectors --trail-name TrailName --region region \
--advanced-event-selectors \
'[
    {
            "Name": "Log IoT Greengrass PutCertificateAuthorities API calls",
            "FieldSelectors": [
                { "Field": "eventCategory", "Equals": ["Data"] },
                { "Field": "resources.type", "Equals": ["AWS::IoT::Thing"] },
                { "Field": "eventName", "Equals": ["PutCertificateAuthorities"] }
            ]
    }
]'
```

Example– Log all Greengrass data events

```
aws cloudtrail put-event-selectors --trail-name TrailName --region region \
--advanced-event-selectors \
'[
    {
        "Name": "Log all certificate data events",
        "FieldSelectors": [
            {
                "Field": "eventCategory",
                "Equals": [
                    "Data"
                ]
            },
            {
                "Field": "resources.type",
                "Equals": [
```

```
                    “AWS::IoT::Certificate”
                ]
            }
        ]
    },
    {
        “Name”: “Log all component version data events”,
        “FieldSelectors”: [
            {
                “Field”: “eventCategory”,
                “Equals”: [
                    “Data”
                ]
            },
            {
                “Field”: “resources.type”,
                “Equals”: [
                    “AWS::GreengrassV2::ComponentVersion”
                ]
            }
        ]
    },
    {
        “Name”: “Log all deployment version”,
        “FieldSelectors”: [
            {
                “Field”: “eventCategory”,
                “Equals”: [
                    “Data”
                ]
            },
            {
                “Field”: “resources.type”,
                “Equals”: [
                    “AWS::GreengrassV2::Deployment”
                ]
            }
        ]
    },
    {
        “Name”: “Log all thing data events”,
        “FieldSelectors”: [
            {
                “Field”: “eventCategory”,
```

```
                “Equals”: [
                    “Data”
                ]
            },
            {
                “Field”: “resources.type”,
                “Equals”: [
                    “AWS::IoT::Thing”
                ]
            }
        ]
    }
]’
```

AWS IoT Greengrass management events in CloudTrail

Management events provide information about management operations that are performed on resources in your AWS account. These are also known as control plane operations. By default, CloudTrail logs management events.

AWS IoT Greengrass logs all AWS IoT Greengrass control plane operations as management events. For a list of the AWS IoT Greengrass control plane operations that AWS IoT Greengrass logs to CloudTrail, see the AWS IoT Greengrass API reference, version 2.

Understanding AWS IoT Greengrass V2 log file entries

A trail is a configuration that enables delivery of events as log files to an S3 bucket that you specify. CloudTrail log files contain one or more log entries. An event represents a single request from any source. It includes information about the requested action, the date and time of the action, request parameters, and so on. CloudTrail log files aren't an ordered stack trace of the public API calls, so they don't appear in any specific order.

The following example shows a CloudTrail log entry that demonstrates the `CreateDeployment` action.

```
{
    "eventVersion": "1.08",
    "userIdentity": {
        "type": "IAMUser",
        "principalId": "AIDACKCEVSQ6C2EXAMPLE",
        "arn": "arn:aws:iam::123456789012:user/Administrator",
        "accountId": "123456789012",
```

```
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "Administrator"
    },
    "eventTime": "2021-01-06T02:38:05Z",
    "eventSource": "greengrass.amazonaws.com",
    "eventName": "CreateDeployment",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "203.0.113.0",
    "userAgent": "aws-cli/2.1.9 Python/3.7.9 Windows/10 exe/AMD64 prompt/off command/
greengrassv2.create-deployment",
    "requestParameters": {
        "deploymentPolicies": {
            "failureHandlingPolicy": "DO_NOTHING",
            "componentUpdatePolicy": {
                "timeoutInSeconds": 60,
                "action": "NOTIFY_COMPONENTS"
            },
            "configurationValidationPolicy": {
                "timeoutInSeconds": 60
            }
        },
        "deploymentName": "Deployment for MyGreengrassCoreGroup",
        "components": {
            "aws.greengrass.Cli": {
                "componentVersion": "2.0.3"
            }
        },
        "iotJobConfiguration": {},
        "targetArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/
MyGreengrassCoreGroup"
    },
    "responseElements": {
        "iotJobArn": "arn:aws:iot:us-west-2:123456789012:job/fdfeba1d-ac6d-44ef-
ab28-54f684ea578d",
        "iotJobId": "fdfeba1d-ac6d-44ef-ab28-54f684ea578d",
        "deploymentId": "4196dddc-0a21-4c54-a985-66a525f6946e"
    },
    "requestID": "311b9529-4aad-42ac-8408-c06c6fec79a9",
    "eventID": "c0f3aa2c-af22-48c1-8161-bad4a2ab1841",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "eventCategory": "Management",
    "recipientAccountId": "123456789012"
```

```
}
```

Gather system health telemetry data from AWS IoT Greengrass core devices

System health telemetry data is diagnostic data that can help you monitor the performance of critical operations on your Greengrass core devices. You can create projects and applications to retrieve, analyze, transform, and report telemetry data from your edge devices. Domain experts, such as process engineers, can use these applications to gain insights into fleet health.

You can use the following methods to gather telemetry data from your Greengrass core devices:

- **Nucleus telemetry emitter component**—The nucleus telemetry emitter component (`aws.greengrass.telemetry.NucleusEmitter`) on a Greengrass core device publishes telemetry data to the `$local/greengrass/telemetry` topic by default. You can use the data that is published to this topic to act locally on your core device, even when your device has limited connectivity to the cloud. Optionally, you can also configure the component to publish telemetry data to an AWS IoT Core MQTT topic of your choice.

  You must deploy the nucleus emitter component to a core device to publish telemetry data. There are no costs associated with publishing telemetry data to the local topic. However, the use of an MQTT topic to publish data to the AWS Cloud is subject to AWS IoT Core pricing.

  AWS IoT Greengrass provides several community components to help you analyze and visualize telemetry data locally on your core device using InfluxDB and Grafana. These components use telemetry data from the nucleus emitter component. For more information, see the README for the InfluxDB publisher component.
- **Telemetry agent**—The telemetry agent on Greengrass core devices collects local telemetry data and publishes it to Amazon EventBridge without requiring any customer interaction. Core devices publish telemetry data to EventBridge on a best effort basis. For example, core devices might fail to deliver telemetry data while offline.

  The telemetry agent feature is enabled by default for all Greengrass core devices. You automatically start to receive data as soon as you set up a Greengrass core device. Aside from your data link costs, the data transfer from the core device to AWS IoT Core is without charge. This is because the agent publishes to an AWS reserved topic. However, depending on your use case, you might incur costs when you receive or process the data.

> **Note**
>
> Amazon EventBridge is an event bus service that you can use to connect your applications with data from a variety of sources, such as Greengrass core devices. For more information, see What is Amazon EventBridge? in the *Amazon EventBridge User Guide*.

To ensure that the the AWS IoT Greengrass Core software functions properly, AWS IoT Greengrass uses the data for development and quality improvement purposes. This feature also helps inform new and enhanced edge capabilities. AWS IoT Greengrass retains telemetry data for up to seven days.

This section describes how to configure and use the telemetry agent. For information about configuring the nucleus telemetry emitter component, see Nucleus telemetry emitter.

Topics

- Telemetry metrics
- Configure telemetry agent settings
- Subscribe to telemetry data in EventBridge

Telemetry metrics

The following table describes the metrics that are published by the telemetry agent.

Name	Description	
System		
`SystemMemUsage`	The amount of memory currently in use by all applications on the Greengras s core device, including the operating system.	
`CpuUsage`	The amount of CPU currently in use by all applications on	

Name	Description
	the Greengrass core device, including the operating system.
TotalNumberOfFDs	The number of file descriptors stored by the operating system of the Greengrass core device. One file descriptor uniquely identifies one open file.
Greengrass nucleus	
NumberOfComponentsRunning	The number of components that are running on the Greengrass core device.
NumberOfComponentsErrored	The number of components that are in error state on the Greengrass core device.
NumberOfComponentsInstalled	The number of components that are installed on the Greengrass core device.
NumberOfComponentsStarting	The number of components that are starting on the Greengrass core device.
NumberOfComponentsNew	The number of components that are new on the Greengrass core device.
NumberOfComponentsStopping	The number of components that are stopping on the Greengrass core device.

Name	Description
NumberOfComponentsFinished	The number of components that are finished on the Greengrass core device.
NumberOfComponentsBroken	The number of components that are broken on the Greengrass core device.
NumberOfComponentsStateless	The number of components that are stateless on the Greengrass core device.
Client device auth – This feature requires v2.4.0 or later of the client device auth component.	
VerifyClientDeviceIdentity.Success	The number of times verifying that the client device identity succeeded.
VerifyClientDeviceIdentity.Failure	The number of times verifying that the client device identity failed.
AuthorizeClientDeviceActions.Success	The number of times the client device is authorized to complete requested actions.
AuthorizeClientDeviceActions.Failure	The number of times the client device is not authorized to complete requested actions.
GetClientDeviceAuthToken.Success	The number of times the client device is successfully authenticated.

Name	Description
`GetClientDeviceAuthToken.Failure`	The number of times the client device is not able to be authenticated.
`SubscribeToCertificateUpdates.Success`	The number of successful subscriptions to certificate updates.
`SubscribeToCertificateUpdates.Failure`	The number of unsuccessful attempts to subscribe to certificate updates.
`ServiceError`	The number of unhandled internal errors across the client device auth.
Stream manager – This feature requires v2.7.0 or later of the Greengrass nucleus component.	
`BytesAppended`	The number of bytes of data appended to stream manager.
`BytesUploadedToIoTAnalytics`	The number of bytes of data that stream manager exports to channels in AWS IoT Analytics.
`BytesUploadedToKinesis`	The number of bytes of data that stream manager exports to streams in Amazon Kinesis Data Streams.

Name	Description
`BytesUploadedToIoTSiteWise`	The number of bytes of data that stream manager exports to asset properties in AWS IoT SiteWise.
`BytesUploadedToS3`	The number of bytes of data that stream manager exports to objects in Amazon S3.
System Metrics – This feature requires v2.15.0 or later of the Greengrass nucleus component.	
`CPUArchitecture`	Device central processing unit architecture.
`Family`	Device operating system family (Windows only).
`KernelVersion`	Device kernel version (Unix only).
`KnowledgeBaseArticles`	Knowledge Base articles installed on a device (Windows only).
`OSBuildMajor`	Major build number of the operating system version (Windows only).
`OSBuildMinor`	Minor build number of the operating system version (Windows only).
`OSName`	Name of the device operating system.

Name	Description	
`OSVersion`	Marketing version of the device operating system.	

Configure telemetry agent settings

The telemetry agent uses the following default settings:

- The telemetry agent aggregates telemetry data every hour.
- The telemetry agent publishes a telemetry message every 24 hours.

The telemetry agent publishes data using the MQTT protocol with a quality of service (QoS) level of 0, which means that it doesn't confirm delivery or retry publishing attempts. Telemetry messages share an MQTT connection with other messages for subscriptions destined for AWS IoT Core.

Aside from your data link costs, the data transfer from the core to AWS IoT Core is without charge. This is because the agent publishes to an AWS reserved topic. However, depending on your use case, you might incur costs when you receive or process the data.

You can enable or disable the telemetry agent feature for each Greengrass core device. You can also configure the intervals over which the core device aggregates and publishes data. To configure telemetry, customize the telemetry configuration parameter when you deploy the Greengrass nucleus component.

Subscribe to telemetry data in EventBridge

You can create rules in Amazon EventBridge that define how to process telemetry data published from the telemetry agent on the Greengrass core device. When EventBridge receives the data, it invokes the target actions defined in your rules. For example, you can create event rules that send notifications, store event information, take corrective action, or invoke other events.

Telemetry events

Telemetry events use the following format.

```
{
```

```
  "version": "0",
  "id": "a09d303e-2f6e-3d3c-a693-8e33f4fe3955",
  "detail-type": "Greengrass Telemetry Data",
  "source": "aws.greengrass",
  "account": "123456789012",
  "time": "2020-11-30T20:45:53Z",
  "region": "us-east-1",
  "resources": [],
  "detail": {
    "ThingName": "MyGreengrassCore",
    "Schema": "2020-07-30",
    "ADP": [
      {
        "TS": 1602186483234,
        "NS": "SystemMetrics",
        "M": [
          {
            "N": "TotalNumberOfFDs",
            "Sum": 6447.0,
            "U": "Count"
          },
          {
            "N": "CpuUsage",
            "Sum": 15.458333333333332,
            "U": "Percent"
          },
          {
            "N": "SystemMemUsage",
            "Sum": 10201.0,
            "U": "Megabytes"
          }
        ]
      },
      {
        "TS": 1602186483234,
        "NS": "GreengrassComponents",
        "M": [
          {
            "N": "NumberOfComponentsStopping",
            "Sum": 0.0,
            "U": "Count"
          },
          {
            "N": "NumberOfComponentsStarting",
```

```
            "Sum": 0.0,
            "U": "Count"
          },
          {
            "N": "NumberOfComponentsBroken",
            "Sum": 0.0,
            "U": "Count"
          },
          {
            "N": "NumberOfComponentsFinished",
            "Sum": 1.0,
            "U": "Count"
          },
          {
            "N": "NumberOfComponentsInstalled",
            "Sum": 0.0,
            "U": "Count"
          },
          {
            "N": "NumberOfComponentsRunning",
            "Sum": 7.0,
            "U": "Count"
          },
          {
            "N": "NumberOfComponentsNew",
            "Sum": 0.0,
            "U": "Count"
          },
          {
            "N": "NumberOfComponentsErrored",
            "Sum": 0.0,
            "U": "Count"
          },
          {
            "N": "NumberOfComponentsStateless",
            "Sum": 0.0,
            "U": "Count"
          }
        ]
      },
      {
        "TS": 1602186483234,
        "NS": "aws.greengrass.ClientDeviceAuth",
        "M": [
```

```
          {
            "N": "VerifyClientDeviceIdentity.Success",
            "Sum": 3.0,
            "U": "Count"
          },
          {
            "N": "VerifyClientDeviceIdentity.Failure",
            "Sum": 1.0,
            "U": "Count"
          },
          {
            "N": "AuthorizeClientDeviceActions.Success",
            "Sum": 20.0,
            "U": "Count"
          },
          {
            "N": "AuthorizeClientDeviceActions.Failure",
            "Sum": 5.0,
            "U": "Count"
          },
          {
            "N": "GetClientDeviceAuthToken.Success",
            "Sum": 5.0,
            "U": "Count"
          },
          {
            "N": "GetClientDeviceAuthToken.Failure",
            "Sum": 2.0,
            "U": "Count"
          },
          {
            "N": "SubscribeToCertificateUpdates.Success",
            "Sum": 10.0,
            "U": "Count"
          },
          {
            "N": "SubscribeToCertificateUpdates.Failure",
            "Sum": 1.0,
            "U": "Count"
          },
          {
            "N": "ServiceError",
            "Sum": 3.0,
            "U": "Count"
```

```
        }
      ]
    },
    {
      "TS": 1602186483234,
      "NS": "aws.greengrass.StreamManager",
      "M": [
        {
          "N": "BytesAppended",
          "Sum": 157745524.0,
          "U": "Bytes"
        },
        {
          "N": "BytesUploadedToIoTAnalytics",
          "Sum": 149012.0,
          "U": "Bytes"
        },
        {
          "N": "BytesUploadedToKinesis",
          "Sum": 12192.0,
          "U": "Bytes"
        },
        {
          "N": "BytesUploadedToIoTSiteWise",
          "Sum": 13321.0,
          "U": "Bytes"
        },
        {
          "N": "BytesUploadedToS3",
          "Sum": 12213.0,
          "U": "Bytes"
        }
      ]
    }
    {
      "TS": 1750104449426,
      "NS": "SystemMetrics",
      "M": [
        {
          "N": "KernelVersion",
          "Sum": 1,
          "U": "6.1.140-154.222.amzn2023.x86_64"
        },
        {
```

```
            "N": "OSVersion",
            "Sum": 1,
            "U": "2023.7.20250609"
          },
          {
            "N": "OSName",
            "Sum": 1,
            "U": "Amazon Linux"
          },
          {
            "N": "CPUArchitecture",
            "Sum": 1,
            "U": "Broadwell (Server)"
          },
        ]
      }
    ]
  }
}
```

The ADP array contains a list of aggregated data points that have the following properties:

TS

The timestamp of when the data was gathered.

NS

The metric namespace.

M

The list of metrics. A metric contains the following properties:

N

The name of the metric.

Sum

The sum of the metric's values in this telemetry event.

U

The unit of the metric value.

For more information about each metric, see Telemetry metrics.

Prerequisites to create EventBridge rules

Before you create an EventBridge rule for AWS IoT Greengrass, you should do the following:

- Familiarize yourself with events, rules, and targets in EventBridge.
- Create and configure the targets invoked by your EventBridge rules. Rules can invoke many types of targets, such as Amazon Kinesis streams, AWS Lambda functions, Amazon SNS topics, and Amazon SQS queues.

  Your EventBridge rule, and the associated targets must be in the AWS Region where you created your Greengrass resources. For more information, see Service endpoints and quotas in the *AWS General Reference*.

For more information, see What is Amazon EventBridge? and Getting started with Amazon EventBridge in the *Amazon EventBridge User Guide*.

Create an event rule to get telemetry data (console)

Use the following steps to use the AWS Management Console to create an EventBridge rule that receives telemetry data published by the Greengrass core device. This allows web servers, email addresses, and other topic subscribers to respond to the event. For more information, see Creating a EventBridge rule that triggers on an event from an AWS resource in the *Amazon EventBridge User Guide*.

1. Open the Amazon EventBridge console, and choose **Create rule**.
2. Under **Name and description**, enter a name and description for the rule.
3. Under **Define pattern**, configure the rule pattern.
   a. Choose **Event pattern**.
   b. Choose **Pre-defined pattern by service**.
   c. For **Service provider**, choose **AWS**.
   d. For **Service name**, choose **Greengrass**.
   e. For **Event type**, select **Greengrass Telemetry Data**.
4. Under **Select event bus**, keep the default event bus options.

5. Under **Select targets**, configure your target. The following example uses an Amazon SQS queue, but you can configure other target types.

   a. For **Target**, choose **SQS queue**.

   b. For **Queue***, choose your target queue.

6. Under **Tags - optional**, define tags for the rule or leave the fields empty.

7. Choose **Create**.

Create an event rule to get telemetry data (CLI)

Use the following steps to use the AWS CLI to create an EventBridge rule that receives telemetry data published by Greengrass core devices. This allows web servers, email addresses, and other topic subscribers to respond to the event.

1. Create the rule.

   - Replace *thing-name* with the thing name of the core device.

   Linux or Unix

   ```
   aws events put-rule \
     --name MyGreengrassTelemetryEventRule \
     --event-pattern "{\"source\": [\"aws.greengrass\"], \"detail\": {\"ThingName
   \": [\"thing-name\"]}}"
   ```

   Windows Command Prompt (CMD)

   ```
   aws events put-rule ^
     --name MyGreengrassTelemetryEventRule ^
     --event-pattern "{\"source\": [\"aws.greengrass\"], \"detail\": {\"ThingName
   \": [\"thing-name\"]}}"
   ```

   PowerShell

   ```
   aws events put-rule `
     --name MyGreengrassTelemetryEventRule `
     --event-pattern "{\"source\": [\"aws.greengrass\"], \"detail\": {\"ThingName
   \": [\"thing-name\"]}}"
   ```

Properties that are omitted from the pattern are ignored.

2. Add the topic as a rule target. The following example uses Amazon SQS but you can configure other target types.

   - Replace *queue-arn* with the ARN of your Amazon SQS queue.

   Linux or Unix

   ```
   aws events put-targets \
     --rule MyGreengrassTelemetryEventRule \
     --targets "Id"="1","Arn"="queue-arn"
   ```

   Windows Command Prompt (CMD)

   ```
   aws events put-targets ^
     --rule MyGreengrassTelemetryEventRule ^
     --targets "Id"="1","Arn"="queue-arn"
   ```

   PowerShell

   ```
   aws events put-targets `
     --rule MyGreengrassTelemetryEventRule `
     --targets "Id"="1","Arn"="queue-arn"
   ```

   Note

   To allow Amazon EventBridge to invoke your target queue, you must add a resource-based policy to your topic. For more information, see Amazon SQS permissions in the *Amazon EventBridge User Guide*.

For more information, see Events and event patterns in EventBridge in the *Amazon EventBridge User Guide*.

Get deployment and component health status notifications

Amazon EventBridge event rules provide you with notifications about state changes for your Greengrass deployments received by your devices and for installed components on your device. EventBridge delivers a near real-time stream of system events that describes changes in AWS resources. AWS IoT Greengrass sends these events to EventBridge on a *best-effort* basis. This means that AWS IoT Greengrass attempts to send all events to EventBridge but, in some rare cases, an event might not be delivered. Additionally, AWS IoT Greengrass might send multiple copies of a given event, which means that your event listeners might not receive the events in the order that the events occurred.

> **Note**
>
> Amazon EventBridge is an event bus service that you can use to connect your applications with data from a variety of sources, such as Greengrass core devices and deployment and component notifications. For more information, see What is Amazon EventBridge? in the *Amazon EventBridge User Guide*.

Topics

- Deployment status change event
- Component status change event
- Prerequisites for creating EventBridge rules
- Configure device health notifications (console)
- Configure device health notifications (CLI)
- Configure device health notifications (CloudFormation)
- See also

Deployment status change event

AWS IoT Greengrass emits an event when a deployment enters the following states: `FAILED`, `SUCCEEDED`, `COMPLETED`, `REJECTED`, and `CANCELED`. You can create an EventBridge rule that runs for all state transitions or transitions to states you specify. When a deployment enters a state that initiates a rule, EventBridge invokes the target actions defined in the rule. This allows you to send

notifications, capture event information, take corrective action, or initiate other events in response to a state change. For example, you can create rules for the following use cases:

- Initiate post-deployment operations, such as downloading assets and notifying personnel.
- Send notifications upon a successful or failed deployment.
- Publish custom metrics about deployment events.

The event for a deployment state change uses the following format:

```
{
    "version":"0",
    "id":" cd4d811e-ab12-322b-8255-EXAMPLEb1bc8",
    "detail-type":"Greengrass V2 Effective Deployment Status Change",
    "source":"aws.greengrass",
    "account":"123456789012",
    "region":"us-west-2",
    "time":"2018-03-22T00:38:11Z",
    "resources":["arn:aws:greengrass:us-
east-1:123456789012:coreDevices:MyGreengrassCore"],
    "detail":{
        "deploymentId": "4f38f1a7-3dd0-42a1-af48-EXAMPLE09681",
        "coreDeviceExecutionStatus": "FAILED|SUCCEEDED|COMPLETED|REJECTED|CANCELED",
        "statusDetails": {
          "errorStack": ["DEPLOYMENT_FAILURE", "ARTIFACT_DOWNLOAD_ERROR", "S3_ERROR",
 "S3_ACCESS_DENIED", "S3_HEAD_OBJECT_ACCESS_DENIED"],
          "errorTypes": ["DEPENDENCY_ERROR", "PERMISSION_ERROR"],
        },
        "reason": "S3_HEAD_OBJECT_ACCESS_DENIED: FAILED_NO_STATE_CHANGE: Failed to
 download artifact name: 's3://pentest27/nucleus/281/aws.greengrass.nucleus.zip' for
 component aws.greengrass.Nucleus-2.8.1, reason: S3 HeadObject returns 403 Access
 Denied. Ensure the IAM role associated with the core device has a policy granting
 s3:GetObject. null (Service: S3, Status Code: 403, Request ID: HR94ZNT2161DAR58,
 Extended Request ID: wTX4DDI+qigQt3uzwl9rlnQiYlBgwvPm/KJFWeFAn9t1mnGXTms/
luLCYANgq08RIH+x2H+hEKc=)"
     }
}
```

You can create rules and events that will update you on the status of a deployment. An event is initiated when a deployment completes as either `FAILED`, `SUCCEEDED`, `COMPLETED`, `REJECTED`, or `CANCELED`. If the deployment failed on the core device, you will receive a detailed response

that explains why the deployment failed. For more information about deployment error codes, see Detailed deployment error codes.

Deployment states

- `FAILED`. The deployment failed.
- `SUCCEEDED`. The deployment targeted to a thing group successfully completed.
- `COMPLETED`. The deployment targeted to a thing successfully completed.
- `REJECTED`. The deployment was rejected. For more information, see the `statusDetails` field.
- `CANCELED`. The deployment was canceled by the user.

It's possible that events might be duplicated or out of order. To determine the order of events, use the `time` property.

For a full list of error codes in `errorStacks` and `errorTypes`, see Detailed deployment error codes and Detailed component status codes.

Component status change event

For AWS IoT Greengrass versions 2.12.2 and earlier, Greengrass emits an event when a component enters the following states: `ERRORED` and `BROKEN`. For Greengrass nucleus versions 2.12.3 and later, Greengrass emits an event when a component enters the following states: `ERRORED`, `BROKEN`, `RUNNING`, and `FINISHED`. Greengrass will also emit an event when a deployment completes. You can create an EventBridge rule that runs for all state transitions or transitions to states you specify. When an installed component enters a state that initiates a rule, EventBridge invokes the target actions defined in the rule. This allows you to send notifications, capture event information, take corrective action, or initiate other events in response to a state change.

The event for a component state change uses the following formats:

Greengrass nucleus v2.12.2 and earlier

<title>Component status: ERRORED or BROKEN</title>

```
{
    "version":"0",
    "id":" cd4d811e-ab12-322b-8255-EXAMPLEb1bc8",
    "detail-type":"Greengrass V2 Installed Component Status Change",
    "source":"aws.greengrass",
```

```
    "account":"123456789012",
    "region":"us-west-2",
    "time":"2018-03-22T00:38:11Z",
    "resources":["arn:aws:greengrass:us-
east-1:123456789012:coreDevices:MyGreengrassCore"],
    "detail": {
       "components": [
          {
            "componentName": "MyComponent",
            "componentVersion": "1.0.0",
            "root": true,
            "lifecycleState": "ERRORED|BROKEN",
            "lifecycleStatusCodes": ["STARTUP_ERROR"],
            "lifecycleStateDetails": "An error occurred during startup. The startup
 script exited with code 1."
          }
        ]
    }
}
```

Greengrass nucleus v2.12.3 and later

<title>Component status: ERRORED or BROKEN</title>

```
{
    "version":"0",
    "id":" cd4d811e-ab12-322b-8255-EXAMPLEb1bc8",
    "detail-type":"Greengrass V2 Installed Component Status Change",
    "source":"aws.greengrass",
    "account":"123456789012",
    "region":"us-west-2",
    "time":"2018-03-22T00:38:11Z",
    "resources":["arn:aws:greengrass:us-
east-1:123456789012:coreDevices:MyGreengrassCore"],
    "detail": {
       "components": [
          {
            "componentName": "MyComponent",
            "componentVersion": "1.0.0",
            "root": true,
            "lifecycleState": "ERRORED|BROKEN",
            "lifecycleStatusCodes": ["STARTUP_ERROR"],
            "lifecycleStateDetails": "An error occurred during startup. The startup
 script exited with code 1."
```

```
          }
        ]
    }
}
```

<title>Component status: RUNNING or FINISHED</title>

```
{
    "version":"0",
    "id":" cd4d811e-ab12-322b-8255-EXAMPLEb1bc8",
    "detail-type":"Greengrass V2 Installed Component Status Change",
    "source":"aws.greengrass",
    "account":"123456789012",
    "region":"us-west-2",
    "time":"2018-03-22T00:38:11Z",
    "resources":["arn:aws:greengrass:us-
east-1:123456789012:coreDevices:MyGreengrassCore"],
    "detail": {
       "components": [
          {
            "componentName": "MyComponent",
            "componentVersion": "1.0.0",
            "root": true,
            "lifecycleState": "RUNNING|FINISHED",
            "lifecycleStateDetails": null
          }
        ]
    }
}
```

You can create rules and events that will update you on the status of an installed component. An event is initiated when a component changes state on the device. You will receive a detailed response that explains why a component is errored or broken. You will also receive a status code that will indicate a reason for the failure. For more information about component status codes, see Detailed component status codes.

Prerequisites for creating EventBridge rules

Before you create an EventBridge rule for AWS IoT Greengrass, do the following:

- Familiarize yourself with events, rules, and targets in EventBridge.

- Create and configure the targets invoked by your EventBridge rules. Rules can invoke many types of targets, including:
  - Amazon Simple Notification Service (Amazon SNS)
  - AWS Lambda functions
  - Amazon Kinesis Video Streams
  - Amazon Simple Queue Service (Amazon SQS) queues

For more information, see What is Amazon EventBridge? and Getting started with Amazon EventBridge in the *Amazon EventBridge User Guide*.

Configure device health notifications (console)

Use the following steps to create an EventBridge rule that publishes an Amazon SNS topic when the deployment state changes for a group. This allows web servers, email addresses, and other topic subscribers to respond to the event. For more information, see Creating a EventBridge rule that triggers on an event from an AWS resource in the *Amazon EventBridge User Guide*.

1. Open the Amazon EventBridge console.
2. In the navigation pane, choose **Rules**.
3. Choose **Create rule**.
4. Enter a name and description for the rule.

   A rule can't have the same name as another rule in the same Region and on the same event bus.
5. For **Event bus**, choose the event bus that you want to associate with this rule. If you want this rule to match events that come from your account, select **AWS default event bus**. When an AWS service in your account emits an event, it always goes to your account's default event bus.
6. For **Rule type**, choose **Rule with an event pattern**.
7. Choose **Next**.
8. For **Event source**, choose **AWS events**.
9. For **Event pattern**, choose **AWS services**.
10. For **AWS service**, choose Greengrass.
11. For **Event type**, choose from the following:

    - For deployment events, choose **Greengrass V2 Effective Deployment Status Change**.

- For component events, choose **Greengrass V2 Installed Component Status Change**.

12. Choose **Next**.
13. For **Target types**, choose **AWS service**.
14. For **Select a target**, configure your target. This example uses an Amazon SNS topic, but you can configure other target types to send notifications.

    a. For **Target**, choose **SNS topic**.

    b. For **Topic**, choose your target topic.

    c. Choose **Next**.
15. Choose **Next**.
16. Review the details of the rule and choose **Create rule**.

Configure device health notifications (CLI)

Use the following steps to create an EventBridge rule that publishes an Amazon SNS topic when there is a Greengrass status change event. This allows web servers, email addresses, and other topic subscribers to respond to the event.

1. Create the rule.

    - For deployment status change events.

    ```
    aws events put-rule \
      --name TestRule \
      --event-pattern "{\"source\": [\"aws.greengrass\"], \"detail-type\":
     [\"Greengrass V2 Effective Deployment Status Change\"]}"
    ```

    - For component status change events.

    ```
    aws events put-rule \
      --name TestRule \
      --event-pattern "{\"source\": [\"aws.greengrass\"], \"detail-type\":
     [\"Greengrass V2 Installed Component Status Change\"]}"
    ```

    Properties that are omitted from the pattern are ignored.
2. Add the topic as a rule target.

- Replace *topic-arn* with the ARN of your Amazon SNS topic.

```
aws events put-targets \
  --rule TestRule \
  --targets "Id"="1","Arn"="topic-arn"
```

> **Note**
>
> To allow Amazon EventBridge to call your target topic, you must add a resource-based policy to your topic. For more information, see Amazon SNS permissions in the *Amazon EventBridge User Guide*.

For more information, see Events and event patterns in EventBridge in the *Amazon EventBridge User Guide*.

Configure device health notifications (CloudFormation)

Use CloudFormation templates to create EventBridge rules that send notifications about state changes for your Greengrass group deployments. For more information, see Amazon EventBridge resource type reference in the *AWS CloudFormation User Guide*.

See also

- Check device deployment status
- What is Amazon EventBridge? in the *Amazon EventBridge User Guide*

Check Greengrass core device status

Greengrass core devices report the status of their software components to AWS IoT Greengrass. You can check the health summary of each device, and you can check the status of each component on each device.

Core devices have the following health statuses:

- HEALTHY – The AWS IoT Greengrass Core software and all components run without issue on the core device.

- UNHEALTHY – The AWS IoT Greengrass Core software or a component is in an error state on the core device.

Note

AWS IoT Greengrass relies on individual devices to send status updates to the AWS Cloud. If the AWS IoT Greengrass Core software isn't running on the device, or if device isn't connected to the AWS Cloud, then the reported status of that device might not reflect its current status. The status timestamp indicates when the device status was last updated. Core devices send status updates at the following times:

- When the AWS IoT Greengrass Core software starts
- When the core device receives a deployment from the AWS Cloud
- For Greengrass nucleus 2.12.2 and earlier, the core device sends status updates when the status of any component on the core device becomes `ERRORED` or `BROKEN`
- For Greengrass nucleus 2.12.3 and later, the core device sends status updates when the status of any component on the core device becomes `ERRORED`, `BROKEN`, `RUNNING`, or `FINISHED`
- At a regular interval that you can configure, which defaults to 24 hours

For AWS IoT Greengrass Core v2.7.0 and later, the core device sends status updates when local deployment and cloud deployment occurs

Topics

- Check health of a core device
- Check health of a core device group
- Check core device component status

Check health of a core device

You can check the status of individual core devices.

To check the status of a core device (AWS CLI)

- Run the following command to retrieve the status of a device. Replace *coreDeviceName* with the name of the core device to query.

  ```
  aws greengrassv2 get-core-device --core-device-thing-name coreDeviceName
  ```

  The response contains information about the core device, including its status.

Check health of a core device group

You can check the status of a group of core devices (a thing group).

To check the status of a group of devices (AWS CLI)

- Run the following command to retrieve the status of multiple core devices. Replace the ARN in the command with the ARN of the thing group to query.

  ```
  aws greengrassv2 list-core-devices --thing-group-arn "arn:aws:iot:region:account-id:thinggroup/thingGroupName"
  ```

  The response contains the list of core devices in the thing group. Each entry in the list contains the status of the core device.

Check core device component status

You can check the status, such as lifecycle state, of the software components on a core device. For more information about component lifecycle states, see Develop AWS IoT Greengrass components.

To check the status of components on a core device (AWS CLI)

- Run the following command to retrieve the status of the components on a core device. Replace *coreDeviceName* with the name of the core device to query.

  ```
  aws greengrassv2 list-installed-components --core-device-thing-name coreDeviceName
  ```

  The response contains the list of components that run on the core device. Each entry in the list contains the lifecycle state of the component, including how current the status of the data is

and when the Greengrass core device last sent a message containing a certain component to the cloud. The response will also include the most recent deployment source that brought the component to the Greengrass core device.

> **Note**
>
> This command retrieves a paginated list of the components that a Greengrass core device runs. By default, this list doesn't include components that are deployed as dependencies of other components. You can include dependencies in the response by setting the `topologyFilter` parameter to `ALL`.

Run AWS Lambda functions

> **Note**
>
> AWS IoT Greengrass doesn't currently support this feature on Windows core devices.

You can import AWS Lambda functions as components that run on AWS IoT Greengrass core devices. You might want to do this in the following cases:

- You have application code in Lambda functions that you want to deploy to core devices.
- You have AWS IoT Greengrass V1 applications that you want to run on AWS IoT Greengrass V2 core devices. For more information, see Step 2: Create and deploy AWS IoT Greengrass V2 components to migrate AWS IoT Greengrass V1 applications.

Lambda functions include dependencies on the following components. You don't need to define these components as dependencies when you import the function. When you deploy the Lambda function component, the deployment includes these Lambda component dependencies.

- The Lambda launcher component (`aws.greengrass.LambdaLauncher`) handles processes and environment configuration.
- The Lambda manager component (`aws.greengrass.LambdaManager`) handles interprocess communication and scaling.
- The Lambda runtimes component (`aws.greengrass.LambdaRuntimes`) provides artifacts for each supported Lambda runtime.

Topics

- Requirements
- Configure Lambda function lifecycle
- Configure Lambda function containerization
- Import a Lambda function as a component (console)
- Import a Lambda function as a component (AWS CLI)

Requirements

Your core devices and Lambda functions must meet the following requirements for you to run the functions on the AWS IoT Greengrass Core software:

- Your core device must meet the requirements to run Lambda functions. If you want the core device to run containerized Lambda functions, the device must meet the requirements to do so. For more information, see Lambda function requirements.
- You must install the programming languages that the Lambda function uses on your core devices.

> **Tip**
>
> You can create a component that installs the programming language, and then specify that component as a dependency of your Lambda function component. Greengrass supports all Lambda supported versions of Python, Node.js, and Java runtimes. Greengrass doesn't apply any additional restrictions on deprecated Lambda runtime versions. You can run Lambda functions that use these deprecated runtimes on AWS IoT Greengrass, but you can't create them in AWS Lambda. For more information about AWS IoT Greengrass support for Lambda runtimes, see Run AWS Lambda functions.

Configure Lambda function lifecycle

The Greengrass Lambda function lifecycle determines when a function starts and how it creates and uses containers. The lifecycle also determines how the AWS IoT Greengrass Core software retains variables and preprocessing logic that are outside of the function handler.

AWS IoT Greengrass supports on-demand (default) and long-lived lifecycles:

- **On-demand** functions start when they are invoked and stop when there are no tasks left to run. Each invocation of the function creates a separate container, also called a sandbox, to process invocations, unless an existing container is available for reuse. Any of the containers might process data that you send to the function.

  Multiple invocations of an on-demand function can run simultaneously.

Variables and preprocessing logic that you define outside of the function handler are not retained when new containers are created.

- **Long-lived** (or *pinned*) functions start when the AWS IoT Greengrass Core software starts and run in a single container. The same container processes all data that you send to the function.

  Multiple invocations are queued until the AWS IoT Greengrass Core software runs earlier invocations.

  Variables and preprocessing logic that you define outside of the function handler are retained for every invocation of the handler.

  Use long-lived Lambda functions when you need to start doing work without any initial input. For example, a long-lived function can load and start processing a machine learning model to be ready when the function receives device data.

  > **Note**
  >
  > Long-lived functions have timeouts that are associated with each invocation of their handler. If you want to invoke code that runs indefinitely, you must start it outside of the handler. Make sure that there's no blocking code outside of the handler that might prevent the function from initializing.
  > These functions run unless the AWS IoT Greengrass Core software stops, such as during a deployment or reboot. These functions won't run if the function encounters an uncaught exception, exceeds its memory limits, or enters an error state, such as a handler timeout.

For more information about container reuse, see Understanding Container Reuse in AWS Lambda in the *AWS Compute Blog*.

Configure Lambda function containerization

By default, Lambda functions run inside of an AWS IoT Greengrass container. Greengrass containers provide isolation between your functions and the host. This isolation increases security for both the host and the functions in the container.

We recommend that you run Lambda functions in a Greengrass container, unless your use case requires them to run without containerization. By running your Lambda functions in a Greengrass container, you have more control over how you restrict access to resources.

You might run a Lambda function without containerization in the following cases:

- You want to run AWS IoT Greengrass on a device that doesn't support container mode. An example would be if you wanted to use a special Linux distribution, or have an earlier kernel version that is out of date.
- You want to run your Lambda function in another container environment with its own OverlayFS, but encounter OverlayFS conflicts when you run in a Greengrass container.
- You need access to local resources with paths that can't be determined at deployment time, or whose paths can change after deployment. An example of this resource would be a pluggable device.
- You have an earlier application that was written as a process, and you encounter issues when you run it in a Greengrass container.

Containerization differences

Containerization	Notes
Greengrass container	• All AWS IoT Greengrass features are available when you run a Lambda function in a Greengrass container. • Lambda functions that run in a Greengrass container don't have access to the deployed code of other Lambda functions, even if they run with the same system group. In other words, your Lambda functions run with increased isolation from one another. • Because the AWS IoT Greengrass Core software runs all child processes in the same container as the Lambda function, the child processes stop when the Lambda function stops.
No container	• The following features aren't available to non-containerized Lambda functions: • Lambda function memory limits.

Containerization	Notes
	• Local device and volume resources. You must access these resources using their file paths on the core device instead of as Lambda function resources. • If your non-containerized Lambda function accesses a machine learning resource, you must identify a resource owner and set access permissions on the resource, not on the Lambda function. • Non-containerized Lambda functions have read-only access to the deployed code of other Lambda functions that run with the same system group.

If you change the containerization for a Lambda function when you deploy it, the function might not work as expected. If the Lambda function uses local resources that are no longer available with the new containerization setting, deployment fails.

- When you change a Lambda function from running in a Greengrass container to running without containerization, the function's memory limits are discarded. You must access the file system directly instead of using attached local resources. You must remove any attached resources before you deploy the Lambda function.
- When you change a Lambda function from running without containerization to running in a container, your Lambda function loses direct access to the file system. You must define a memory limit for each function or accept the default 16 MB memory limit. You can configure these settings for each Lambda function when you deploy it.

To change containerization settings for a Lambda function component, set the value of the `containerMode` configuration parameter to one of the following options when you deploy the component.

- `NoContainer` – The component doesn't run in an isolated runtime environment.

- `GreengrassContainer` – The component runs in an isolated runtime environment inside the AWS IoT Greengrass container.

For more information about how to deploy and configure components, see Deploy AWS IoT Greengrass components to devices and Update component configurations.

Import a Lambda function as a component (console)

When you use the AWS IoT Greengrass console to create a Lambda function component, you import an existing AWS Lambda function and then configure it to create a component that runs on your Greengrass device.

Before you begin, review the requirements to run Lambda functions on Greengrass devices.

Tasks

- Step 1: Choose a Lambda function to import
- Step 2: Configure Lambda function parameters
- Step 3: (Optional) Specify supported platforms for the Lambda function
- Step 4: (Optional) Specify component dependencies for the Lambda function
- Step 5: (Optional) Run the Lambda function in a container
- Step 6: Create the Lambda function component

Step 1: Choose a Lambda function to import

1. In the AWS IoT Greengrass console navigation menu, choose **Components**.
2. On the **Components** page, choose **Create component**.
3. On the **Create component** page, under **Component information**, choose **Import Lambda function**.
4. In **Lambda function**, search for and choose the Lambda function that you want to import.

   AWS IoT Greengrass creates the component with the name of the Lambda function.
5. In **Lambda function version**, choose the version to import. You can't choose Lambda aliases like `$LATEST`.

AWS IoT Greengrass creates the component with the version of the Lambda function as a valid semantic version. For example, if your function version is 3, the component version becomes `3.0.0`.

Step 2: Configure Lambda function parameters

On the **Create component** page, under **Lambda function configuration**, configure the following parameters to use to run the Lambda function.

1. (Optional) Add the list of event sources to which the Lambda function subscribes for work messages. You can specify event sources to subscribe this function to local publish/subscribe messages and AWS IoT Core MQTT messages. The Lambda function is called when it receives a message from an event source.

   > **Note**
   >
   > To subscribe this function to messages from other Lambda functions or components, deploy the legacy subscription router component when you deploy this Lambda function component. When you deploy the legacy subscription router component, specify the subscriptions that the Lambda function uses.

   Under **Event sources**, do the following to add an event source:

   a. For each event source that you add, specify the following options:

      - **Topic** – The topic to subscribe for messages.
      - **Type** – The type of event source. Choose from the following options:
        - **Local publish/subscribe** – Subscribe to local publish/subscribe messages.

          If you use Greengrass nucleus v2.6.0 or later and Lambda manager v2.2.5 or later, you can use MQTT topic wildcards (+ and #) in the **Topic** when you specify this type.
        - **AWS IoT Core MQTT** – Subscribe to AWS IoT Core MQTT messages.

          You can use MQTT topic wildcards (+ and #) in the **Topic** when you specify this type.

   b. To add another event source, choose **Add event source** and repeat the previous step. To remove an event source, choose **Remove** next to the event source that you want to remove.

2. For **Timeout (seconds)**, enter the maximum amount of time in seconds that a non-pinned Lambda function can run before it times out. The default is 3 seconds.
3. For **Pinned**, choose whether the Lambda function component is pinned. The default is **True**.

   - A pinned (or long-lived) Lambda function starts when AWS IoT Greengrass starts and keeps running in its own container.
   - A non-pinned (or on-demand) Lambda function starts only when it receives a work item and exits after it remains idle for a specified maximum idle time. If the function has multiple work items, the AWS IoT Greengrass Core software creates multiple instances of the function.

4. (Optional) Under **Additional parameters**, set the following Lambda function parameters.

   - **Status timeout (seconds)** – The interval in seconds at which the Lambda function component sends status updates to the Lambda manager component. This parameter applies only to pinned functions. The default is 60 seconds.
   - **Maximum queue size** – The maximum size of the message queue for the Lambda function component. The AWS IoT Greengrass Core software stores messages in a FIFO (first-in, first-out) queue until it can run the Lambda function to consume each message. The default is 1,000 messages.
   - **Maximum number of instances** – The maximum number of instances that a non-pinned Lambda function can run at the same time. The default is 100 instances.
   - **Maximum idle time (seconds)** – The maximum amount of time in seconds that a non-pinned Lambda function can idle before the AWS IoT Greengrass Core software stops its process. The default is 60 seconds.
   - **Encoding type** – The type of payload that the Lambda function supports. Choose from the following options:
     - **JSON**
     - **Binary**

     The default is JSON.

5. (Optional) Specify the list of command line arguments to pass to the Lambda function when it runs.

a. Under **Additional parameters, Process arguments**, choose **Add argument**.

b. For each argument that you add, enter the argument that you want to pass to the function.

c. To remove an argument, choose **Remove** next to the argument that you want to remove.

6. (Optional) Specify the environment variables that are available to the Lambda function when it runs. Environment variables enable you to store and update configuration settings without the need to change function code.

   a. Under **Additional parameters, Environment variables**, choose **Add environment variable**.

   b. For each environment variable that you add, specify the following options:

      - **Key** – The variable name.
      - **Value** – The default value for this variable.

   c. To remove an environment variable, choose **Remove** next to the environment variable that you want to remove.

Step 3: (Optional) Specify supported platforms for the Lambda function

All core devices have attributes for operating system and architecture. When you deploy the Lambda function component, the AWS IoT Greengrass Core software compares the platform values that you specify with the platform attributes on the core device to determine whether the Lambda function is supported on that device.

> **Note**
>
> You can also specify custom platform attributes when you deploy the Greengrass nucleus component to a core device. For more information, see the platform overrides parameter of the Greengrass nucleus component.

Under **Lambda function configuration, Additional parameters, Platforms**, do the following to specify the platforms that this Lambda function supports.

1. For each platform, specify the following options:

- **Operating system** – The name of the operating system for the platform. Currently, the only supported value is `linux`.
- **Architecture** – The processor architecture for the platform. Supported values are:
  - `amd64`
  - `arm`
  - `aarch64`
  - `x86`

2. To add another platform, choose **Add platform** and repeat the previous step. To remove a supported platform, choose **Remove** next to the platform that you want to remove.

Step 4: (Optional) Specify component dependencies for the Lambda function

Component dependencies identify additional AWS-provided components or custom components that your function uses. When you deploy the Lambda function component, the deployment includes these dependencies for your function to run.

> **Important**
>
> To import a Lambda function that you created to run on AWS IoT Greengrass V1, you must define individual component dependencies for the features that your function uses, such as secrets, local shadows, and stream manager. Define these components as hard dependencies so that your Lambda function component restarts if the dependency changes state. For more information, see Import V1 Lambda functions.

Under **Lambda function configuration, Additional parameters, Component dependencies**, complete the following steps to specify the component dependencies for your Lambda function.

1. Choose **Add dependency**.
2. For each component dependency that you add, specify the following options:
   - **Component name** – The component name. For example, enter **`aws.greengrass.StreamManager`** to include the stream manager component.

- **Version requirement** – The npm-style semantic version constraint that identifies the compatible versions of this component dependency. You can specify a single version or a range of versions. For example, enter **`^1.0.0`** to specify that this Lambda function depends on any version in the first major version of the stream manager component. For more information about semantic version constraints, see the npm semver calculator.
- **Type** – The type of dependency. Choose from the following options:
  - **Hard** – The Lambda function component restarts if the dependency changes state. This is the default selection.
  - **Soft** – The Lambda function component doesn't restart if the dependency changes state.

3. To remove a component dependency, choose **Remove** next to the component dependency

Step 5: (Optional) Run the Lambda function in a container

By default, Lambda functions run in an isolated runtime environment inside the AWS IoT Greengrass Core software. You can also choose to run the Lambda function as a process without any isolation (that is, in **No container** mode).

Under **Linux process configuration**, for **Isolation mode**, choose from the following options to select the containerization for your Lambda function:

- **Greengrass container** – The Lambda function runs in a container. This is the default selection.
- **No container** – The Lambda function runs as a process without any isolation.

If you run the Lambda function in a container, complete the following steps to configure the process configuration for the Lambda function.

1. Configure the amount of memory and the system resources, such as volumes and devices, to make available to the container.

   Under **Container parameters**, do the following.

   a. For **Memory size**, enter the memory size that you want to allocate to the container. You can specify the memory size in **MB** or **kB**.

   b. For **Read-only sys folder**, choose whether or not the container can read information from the device's `/sys` folder. The default is **False**.

2. (Optional) Configure the local volumes that the containerized Lambda function can access. When you define a volume, the AWS IoT Greengrass Core software mounts the source files to the destination inside the container.

   a. Under **Volumes**, choose **Add volume**.

   b. For each volume that you add, specify the following options:

      - **Physical volume** – The path to the source folder on the core device.
      - **Logical volume** – The path to the destination folder in the container.
      - **Permission** – (Optional) The permission to access the source folder from the container. Choose from the following options:
        - **Read-only** – The Lambda function has read-only access to the source folder. This is the default selection.
        - **Read-write** – The Lambda function has read/write access to the source folder.
      - **Add group owner** – (Optional) Whether or not to add the system group that runs the Lambda function component as an owner of the source folder. The default is **False**.

   c. To remove a volume, choose **Remove** next to the volume that you want to remove.

3. (Optional) Configure the local system devices that the containerized Lambda function can access.

   a. Under **Devices**, choose **Add device**.

   b. For each device that you add, specify the following options:

      - **Mount path** – The path to the system device on the core device.
      - **Permission** – (Optional) The permission to access the system device from the container. Choose from the following options:
        - **Read-only** – The Lambda function has read-only access to the system device. This is the default selection.
        - **Read-write** – The Lambda function has read/write access to the source folder.
      - **Add group owner** – (Optional) Whether or not to add the system group that runs the Lambda function component as an owner of the system device. The default is **False**.

Step 6: Create the Lambda function component

After you configure settings for your Lambda function component, choose **Create** to finish creating the new component.

To run the Lambda function on your core device, you can then deploy the new component to your core devices. For more information, see Deploy AWS IoT Greengrass components to devices.

Import a Lambda function as a component (AWS CLI)

Use the CreateComponentVersion operation to create components from Lambda functions. When you call this operation, specify `lambdaFunction` to import a Lambda function.

Tasks

- Step 1: Define the Lambda function configuration
- Step 2: Create the Lambda function component

Step 1: Define the Lambda function configuration

1. Create a file called `lambda-function-component.json`, and then copy the following JSON object into the file. Replace the `lambdaArn` with the ARN of the Lambda function to import.

   ```
   {
     "lambdaFunction": {
       "lambdaArn": "arn:aws:lambda:region:account-id:function:HelloWorld:1"
     }
   }
   ```

   > **Important**
   >
   > You must specify an ARN that includes the version of the function to import. You can't use version aliases like `$LATEST`.

2. (Optional) Specify the name (`componentName`) of the component. If you omit this parameter, AWS IoT Greengrass creates the component with the name of the Lambda function.

   ```
   {
     "lambdaFunction": {
   ```

```
    "lambdaArn": "arn:aws:lambda:region:account-id:function:HelloWorld:1",
    "componentName": "com.example.HelloWorldLambda"
  }
}
```

3. (Optional) Specify the version (`componentVersion`) for the component. If you omit this parameter, AWS IoT Greengrass creates the component with the version of the Lambda function as a valid semantic version. For example, if your function version is `3`, the component version becomes `3.0.0`.

   > **Note**
   >
   > Each component version that you upload must be unique. Make sure that you upload the correct component version, because you can't edit it after you upload it.
   > AWS IoT Greengrass uses semantic versions for components. Semantic versions follow a *major.minor.patch* number system. For example, version `1.0.0` represents the first major release for a component. For more information, see the semantic version specification.

   ```
   {
     "lambdaFunction": {
       "lambdaArn": "arn:aws:lambda:region:account-id:function:HelloWorld:1",
       "componentName": "com.example.HelloWorldLambda",
       "componentVersion": "1.0.0"
     }
   }
   ```

4. (Optional) Specify the platforms that this Lambda function supports. Each platform contains a map of attributes that identify a platform. All core devices have attributes for operating system (`os`) and architecture (`architecture`). The AWS IoT Greengrass Core software may add other platform attributes. You can also specify custom platform attributes when you deploy the Greengrass nucleus component to a core device. Do the following:

   a. Add a list of platforms (`componentPlatforms`) to the Lambda function in `lambda-function-component.json`.

      ```
      {
        "lambdaFunction": {
          "lambdaArn": "arn:aws:lambda:region:account-id:function:HelloWorld:1",
      ```

```
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [

    ]
  }
}
```

b. Add each supported platform to the list. Each platform has a friendly `name` to identify it and a map of attributes. The following example specifies that this function supports x86 devices that run Linux.

```
{
  "name": "Linux x86",
  "attributes": {
    "os": "linux",
    "architecture": "x86"
  }
}
```

Your `lambda-function-component.json` might contain a document similar to the following example.

```
{
  "lambdaFunction": {
    "lambdaArn": "arn:aws:lambda:region:account-id:function:HelloWorld:1",
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [
      {
        "name": "Linux x86",
        "attributes": {
          "os": "linux",
          "architecture": "x86"
        }
      }
    ]
  }
}
```

5. (Optional) Specify the component dependencies for your Lambda function. When you deploy the Lambda function component, the deployment includes these dependencies for your function to run.

   > **Important**
   >
   > To import a Lambda function that you created to run on AWS IoT Greengrass V1, you must define individual component dependencies for the features that your function uses, such as secrets, local shadows, and stream manager. Define these components as hard dependencies so that your Lambda function component restarts if the dependency changes state. For more information, see Import V1 Lambda functions.

   Do the following:

   a. Add a map of component dependencies (`componentDependencies`) to the Lambda function in `lambda-function-component.json`.

   ```
   {
     "lambdaFunction": {
       "lambdaArn": "arn:aws:lambda:region:account-id:function:HelloWorld:1",
       "componentName": "com.example.HelloWorldLambda",
       "componentVersion": "1.0.0",
       "componentPlatforms": [
         {
           "name": "Linux x86",
           "attributes": {
             "os": "linux",
             "architecture": "x86"
           }
         }
       ],
       "componentDependencies": {

       }
     }
   }
   ```

   b. Add each component dependency to the map. Specify the component name as the key and specify an object with the following parameters:

- `versionRequirement` – The npm-style semantic version constraint that identifies the compatible versions of the component dependency. You can specify a single version or a range of versions. For more information about semantic version constraints, see the npm semver calculator.
- `dependencyType` – (Optional) The type of the dependency. Choose from the following:
  - `SOFT` – The Lambda function component doesn't restart if the dependency changes state.
  - `HARD` – The Lambda function component restarts if the dependency changes state.

  The default is `HARD`.

The following example specifies that this Lambda function depends on any version in the first major version of the stream manager component. The Lambda function component restarts when stream manager restarts or updates.

```
{
  "aws.greengrass.StreamManager": {
    "versionRequirement": "^1.0.0",
    "dependencyType": "HARD"
  }
}
```

Your `lambda-function-component.json` might contain a document similar to the following example.

```
{
  "lambdaFunction": {
    "lambdaArn": "arn:aws:lambda:region:account-id:function:HelloWorld:1",
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [
      {
        "name": "Linux x86",
        "attributes": {
          "os": "linux",
          "architecture": "x86"
        }
      }
```

```
    ],
    "componentDependencies": {
      "aws.greengrass.StreamManager": {
        "versionRequirement": "^1.0.0",
        "dependencyType": "HARD"
      }
    }
  }
}
```

6. (Optional) Configure the Lambda function parameters to use to run the function. You can configure options such environment variables, message event sources, timeouts, and container settings. Do the following:

   a. Add the Lambda parameters object (`componentLambdaParameters`) to the Lambda function in `lambda-function-component.json`.

```
{
  "lambdaFunction": {
    "lambdaArn": "arn:aws:lambda:region:account-id:function:HelloWorld:1",
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [
      {
        "name": "Linux x86",
        "attributes": {
          "os": "linux",
          "architecture": "x86"
        }
      }
    ],
    "componentDependencies": {
      "aws.greengrass.StreamManager": {
        "versionRequirement": "^1.0.0",
        "dependencyType": "HARD"
      }
    },
    "componentLambdaParameters": {

    }
  }
}
```

b. (Optional) Specify the event sources to which the Lambda function subscribes for work messages. You can specify event sources to subscribe this function to local publish/subscribe messages and AWS IoT Core MQTT messages. The Lambda function is called when it receives a message from an event source.

> **Note**
>
> To subscribe this function to messages from other Lambda functions or components, deploy the legacy subscription router component when you deploy this Lambda function component. When you deploy the legacy subscription router component, specify the subscriptions that the Lambda function uses.

Do the following:

i. Add the list of event sources (`eventSources`) to the Lambda function parameters.

```
{
  "lambdaFunction": {
    "lambdaArn": "arn:aws:lambda:region:account-id:function:HelloWorld:1",
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [
      {
        "name": "Linux x86",
        "attributes": {
          "os": "linux",
          "architecture": "x86"
        }
      }
    ],
    "componentDependencies": {
      "aws.greengrass.StreamManager": {
        "versionRequirement": "^1.0.0",
        "dependencyType": "HARD"
      }
    },
    "componentLambdaParameters": {
      "eventSources": [

      ]
```

```
    }
  }
}
```

ii. Add each event source to the list. Each event source has the following parameters:

- `topic` – The topic to subscribe for messages.
- `type` – The type of event source. Choose from the following options:
  - PUB_SUB – Subscribe to local publish/subscribe messages.

    If you use Greengrass nucleus v2.6.0 or later and Lambda manager v2.2.5 or later, you can use MQTT topic wildcards (+ and #) in the `topic` when you specify this type.
  - `IOT_CORE` – Subscribe to AWS IoT Core MQTT messages.

    You can use MQTT topic wildcards (+ and #) in the `topic` when you specify this type.

The following example subscribes to AWS IoT Core MQTT on topics that match the `hello/world/+` topic filter.

```
{
  "topic": "hello/world/+",
  "type": "IOT_CORE"
}
```

Your `lambda-function-component.json` might look similar to the following example.

```
{
  "lambdaFunction": {
    "lambdaArn": "arn:aws:lambda:region:account-
id:function:HelloWorld:1",
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [
      {
        "name": "Linux x86",
        "attributes": {
          "os": "linux",
```

```
          "architecture": "x86"
        }
      }
    ],
    "componentDependencies": {
      "aws.greengrass.StreamManager": {
        "versionRequirement": "^1.0.0",
        "dependencyType": "HARD"
      }
    },
    "componentLambdaParameters": {
      "eventSources": [
        {
          "topic": "hello/world/+",
          "type": "IOT_CORE"
        }
      ]
    }
  }
}
```

c. (Optional) Specify any of the following parameters in the Lambda function parameters object:

- `environmentVariables` – The map of environment variables that are available to the Lambda function when it runs.
- `execArgs` – The list of arguments to pass to the Lambda function when it runs.
- `inputPayloadEncodingType` – The type of payload that the Lambda function supports. Choose from the following options:
  - `json`
  - `binary`

  Default: `json`
- `pinned` – Whether or not the Lambda function is pinned. The default is `true`.
  - A pinned (or long-lived) Lambda function starts when AWS IoT Greengrass starts and keeps running in its own container.
  - A non-pinned (or on-demand) Lambda function starts only when it receives a work item and exits after it remains idle for a specified maximum idle time. If the function

has multiple work items, the AWS IoT Greengrass Core software creates multiple instances of the function.

Use `maxIdleTimeInSeconds` to set the maximum idle time for your function.

- `timeoutInSeconds` – The maximum amount of time in seconds that the Lambda function can run before it times out. The default is 3 seconds.
- `statusTimeoutInSeconds` – The interval in seconds at which the Lambda function component sends status updates to the Lambda manager component. This parameter applies only to pinned functions. The default is 60 seconds.
- `maxIdleTimeInSeconds` – The maximum amount of time in seconds that a non-pinned Lambda function can idle before the AWS IoT Greengrass Core software stops its process. The default is 60 seconds.
- `maxInstancesCount` – The maximum number of instances that a non-pinned Lambda function can run at the same time. The default is 100 instances.
- `maxQueueSize` – The maximum size of the message queue for the Lambda function component. The AWS IoT Greengrass Core software stores messages in a FIFO (first-in-first-out) queue until it can run the Lambda function to consume each message. The default is 1,000 messages.

Your `lambda-function-component.json` might contain a document similar to the following example.

```
{
  "lambdaFunction": {
    "lambdaArn": "arn:aws:lambda:region:account-id:function:HelloWorld:1",
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [
      {
        "name": "Linux x86",
        "attributes": {
          "os": "linux",
          "architecture": "x86"
        }
      }
    ],
    "componentDependencies": {
      "aws.greengrass.StreamManager": {
```

```
        "versionRequirement": "^1.0.0",
        "dependencyType": "HARD"
      }
    },
    "componentLambdaParameters": {
      "eventSources": [
        {
          "topic": "hello/world/+",
          "type": "IOT_CORE"
        }
      ],
      "environmentVariables": {
        "LIMIT": "300"
      },
      "execArgs": [
        "-d"
      ],
      "inputPayloadEncodingType": "json",
      "pinned": true,
      "timeoutInSeconds": 120,
      "statusTimeoutInSeconds": 30,
      "maxIdleTimeInSeconds": 30,
      "maxInstancesCount": 50,
      "maxQueueSize": 500
    }
  }
}
```

d. (Optional) Configure the container settings for the Lambda function. By default, Lambda functions run in an isolated runtime environment inside the AWS IoT Greengrass Core software. You can also choose to run the Lambda function as a process without any isolation. If you run the Lambda function in a container, you configure the memory size of the container and what system resources are available to the Lambda function. Do the following:

   i. Add the Linux process parameters object (`linuxProcessParams`) to the Lambda parameters object in `lambda-function-component.json`.

```
{
  "lambdaFunction": {
    "lambdaArn": "arn:aws:lambda:region:account-id:function:HelloWorld:1",
    "componentName": "com.example.HelloWorldLambda",
```

```
    "componentVersion": "1.0.0",
    "componentPlatforms": [
      {
        "name": "Linux x86",
        "attributes": {
          "os": "linux",
          "architecture": "x86"
        }
      }
    ],
    "componentDependencies": {
      "aws.greengrass.StreamManager": {
        "versionRequirement": "^1.0.0",
        "dependencyType": "HARD"
      }
    },
    "componentLambdaParameters": {
      "eventSources": [
        {
          "topic": "hello/world/+",
          "type": "IOT_CORE"
        }
      ],
      "environmentVariables": {
        "LIMIT": "300"
      },
      "execArgs": [
        "-d"
      ],
      "inputPayloadEncodingType": "json",
      "pinned": true,
      "timeoutInSeconds": 120,
      "statusTimeoutInSeconds": 30,
      "maxIdleTimeInSeconds": 30,
      "maxInstancesCount": 50,
      "maxQueueSize": 500,
      "linuxProcessParams": {

      }
    }
  }
}
```

ii. (Optional) Specify whether or not the Lambda function runs in a container. Add the `isolationMode` parameter to the process parameters object, and choose from the following options:

- `GreengrassContainer` – The Lambda function runs in a container.
- `NoContainer` – The Lambda function runs as a process without any isolation.

The default is `GreengrassContainer`.

iii. (Optional) If you run the Lambda function in a container, you can configure the amount of memory and the system resources, such as volumes and devices, to make available to the container. Do the following:

A. Add the container parameters object (`containerParams`) to the Linux process parameters object in `lambda-function-component.json`.

```
{
  "lambdaFunction": {
    "lambdaArn": "arn:aws:lambda:region:account-
id:function:HelloWorld:1",
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [
      {
        "name": "Linux x86",
        "attributes": {
          "os": "linux",
          "architecture": "x86"
        }
      }
    ],
    "componentDependencies": {
      "aws.greengrass.StreamManager": {
        "versionRequirement": "^1.0.0",
        "dependencyType": "HARD"
      }
    },
    "componentLambdaParameters": {
      "eventSources": [
        {
          "topic": "hello/world/+",
```

```
          "type": "IOT_CORE"
        }
      ],
      "environmentVariables": {
        "LIMIT": "300"
      },
      "execArgs": [
        "-d"
      ],
      "inputPayloadEncodingType": "json",
      "pinned": true,
      "timeoutInSeconds": 120,
      "statusTimeoutInSeconds": 30,
      "maxIdleTimeInSeconds": 30,
      "maxInstancesCount": 50,
      "maxQueueSize": 500,
      "linuxProcessParams": {
        "containerParams": {

        }
      }
    }
  }
}
```

B. (Optional) Add the `memorySizeInKB` parameter to specify the memory size of the container. The default is 16,384 KB (16 MB).

C. (Optional) Add the `mountROSysfs` parameter to specify whether or not the container can read information from the device's `/sys` folder. The default is `false`.

D. (Optional) Configure the local volumes that the containerized Lambda function can access. When you define a volume, the AWS IoT Greengrass Core software mounts the source files to the destination inside the container. Do the following:

   I. Add the list of volumes (`volumes`) to the container parameters.

```
{
  "lambdaFunction": {
    "lambdaArn": "arn:aws:lambda:region:account-
id:function:HelloWorld:1",
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
```

```
    "componentPlatforms": [
      {
        "name": "Linux x86",
        "attributes": {
          "os": "linux",
          "architecture": "x86"
        }
      }
    ],
    "componentDependencies": {
      "aws.greengrass.StreamManager": {
        "versionRequirement": "^1.0.0",
        "dependencyType": "HARD"
      }
    },
    "componentLambdaParameters": {
      "eventSources": [
        {
          "topic": "hello/world/+",
          "type": "IOT_CORE"
        }
      ],
      "environmentVariables": {
        "LIMIT": "300"
      },
      "execArgs": [
        "-d"
      ],
      "inputPayloadEncodingType": "json",
      "pinned": true,
      "timeoutInSeconds": 120,
      "statusTimeoutInSeconds": 30,
      "maxIdleTimeInSeconds": 30,
      "maxInstancesCount": 50,
      "maxQueueSize": 500,
      "linuxProcessParams": {
        "containerParams": {
          "memorySizeInKB": 32768,
          "mountROSysfs": true,
          "volumes": [

          ]
        }
      }
```

```
    }
  }
}
```

II. Add each volume to the list. Each volume has the following parameters:

- `sourcePath` – The path to the source folder on the core device.
- `destinationPath` – The path to the destination folder in the container.
- `permission` – (Optional) The permission to access the source folder from the container. Choose from the following options:
  - `ro` – The Lambda function has read-only access to the source folder.
  - `rw` – The Lambda function has read-write access to the source folder.

  The default is `ro`.
- `addGroupOwner` – (Optional) Whether or not to add the system group that runs the Lambda function component as an owner of the source folder. The default is `false`.

Your `lambda-function-component.json` might contain a document similar to the following example.

```
{
  "lambdaFunction": {
    "lambdaArn": "arn:aws:lambda:region:account-
id:function:HelloWorld:1",
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [
      {
        "name": "Linux x86",
        "attributes": {
          "os": "linux",
          "architecture": "x86"
        }
      }
    ],
    "componentDependencies": {
      "aws.greengrass.StreamManager": {
        "versionRequirement": "^1.0.0",
        "dependencyType": "HARD"
```

```
      }
    },
    "componentLambdaParameters": {
      "eventSources": [
        {
          "topic": "hello/world/+",
          "type": "IOT_CORE"
        }
      ],
      "environmentVariables": {
        "LIMIT": "300"
      },
      "execArgs": [
        "-d"
      ],
      "inputPayloadEncodingType": "json",
      "pinned": true,
      "timeoutInSeconds": 120,
      "statusTimeoutInSeconds": 30,
      "maxIdleTimeInSeconds": 30,
      "maxInstancesCount": 50,
      "maxQueueSize": 500,
      "linuxProcessParams": {
        "containerParams": {
          "memorySizeInKB": 32768,
          "mountROSysfs": true,
          "volumes": [
            {
              "sourcePath": "/var/data/src",
              "destinationPath": "/var/data/dest",
              "permission": "rw",
              "addGroupOwner": true
            }
          ]
        }
      }
    }
  }
}
```

E. (Optional) Configure the local system devices that the containerized Lambda function can access. Do the following:

   I. Add the list of system devices (`devices`) to the container parameters.

```
{
  "lambdaFunction": {
    "lambdaArn": "arn:aws:lambda:region:account-
id:function:HelloWorld:1",
    "componentName": "com.example.HelloWorldLambda",
    "componentVersion": "1.0.0",
    "componentPlatforms": [
      {
        "name": "Linux x86",
        "attributes": {
          "os": "linux",
          "architecture": "x86"
        }
      }
    ],
    "componentDependencies": {
      "aws.greengrass.StreamManager": {
        "versionRequirement": "^1.0.0",
        "dependencyType": "HARD"
      }
    },
    "componentLambdaParameters": {
      "eventSources": [
        {
          "topic": "hello/world/+",
          "type": "IOT_CORE"
        }
      ],
      "environmentVariables": {
        "LIMIT": "300"
      },
      "execArgs": [
        "-d"
      ],
      "inputPayloadEncodingType": "json",
      "pinned": true,
      "timeoutInSeconds": 120,
      "statusTimeoutInSeconds": 30,
      "maxIdleTimeInSeconds": 30,
      "maxInstancesCount": 50,
      "maxQueueSize": 500,
      "linuxProcessParams": {
        "containerParams": {
```

```
          "memorySizeInKB": 32768,
          "mountROSysfs": true,
          "volumes": [
            {
              "sourcePath": "/var/data/src",
              "destinationPath": "/var/data/dest",
              "permission": "rw",
              "addGroupOwner": true
            }
          ],
          "devices": [

          ]
        }
      }
    }
  }
}
```

II. Add each system device to the list. Each system device has the following parameters:

- `path` – The path to the system device on the core device.
- `permission` – (Optional) The permission to access the system device from the container. Choose from the following options:
  - `ro` – The Lambda function has read-only access to the system device.
  - `rw` – The Lambda function has read-write access to the system device.

  The default is `ro`.
- `addGroupOwner` – (Optional) Whether or not to add the system group that runs the Lambda function component as an owner of the system device. The default is `false`.

Your `lambda-function-component.json` might contain a document similar to the following example.

```
{
  "lambdaFunction": {
    "lambdaArn": "arn:aws:lambda:region:account-
id:function:HelloWorld:1",
```

```
  "componentName": "com.example.HelloWorldLambda",
  "componentVersion": "1.0.0",
  "componentPlatforms": [
    {
      "name": "Linux x86",
      "attributes": {
        "os": "linux",
        "architecture": "x86"
      }
    }
  ],
  "componentDependencies": {
    "aws.greengrass.StreamManager": {
      "versionRequirement": "^1.0.0",
      "dependencyType": "HARD"
    }
  },
  "componentLambdaParameters": {
    "eventSources": [
      {
        "topic": "hello/world/+",
        "type": "IOT_CORE"
      }
    ],
    "environmentVariables": {
      "LIMIT": "300"
    },
    "execArgs": [
      "-d"
    ],
    "inputPayloadEncodingType": "json",
    "pinned": true,
    "timeoutInSeconds": 120,
    "statusTimeoutInSeconds": 30,
    "maxIdleTimeInSeconds": 30,
    "maxInstancesCount": 50,
    "maxQueueSize": 500,
    "linuxProcessParams": {
      "containerParams": {
        "memorySizeInKB": 32768,
        "mountROSysfs": true,
        "volumes": [
          {
            "sourcePath": "/var/data/src",
```

```
              "destinationPath": "/var/data/dest",
              "permission": "rw",
              "addGroupOwner": true
            }
          ],
          "devices": [
            {
              "path": "/dev/sda3",
              "permission": "rw",
              "addGroupOwner": true
            }
          ]
        }
      }
    }
  }
}
```

7. (Optional) Add tags (`tags`) for the component. For more information, see Tag your AWS IoT Greengrass Version 2 resources.

Step 2: Create the Lambda function component

1. Run the following command to create the Lambda function component from `lambda-function-component.json`.

```
aws greengrassv2 create-component-version --cli-input-json file://lambda-function-component.json
```

The response looks similar to the following example if the request succeeds.

```
{
  "arn":
 "arn:aws:greengrass:region:123456789012:components:com.example.HelloWorldLambda:versions:1
  "componentName": "com.example.HelloWorldLambda",
  "componentVersion": "1.0.0",
  "creationTimestamp": "Mon Dec 15 20:56:34 UTC 2020",
  "status": {
    "componentState": "REQUESTED",
    "message": "NONE",
    "errors": {}
```

```
  }
}
```

Copy the `arn` from the output to check the state of the component in the next step.

2. When you create a component, its state is `REQUESTED`. Then, AWS IoT Greengrass validates that the component is deployable. You can run the following command to query the component status and verify that your component is deployable. Replace the `arn` with the ARN from the previous step.

```
aws greengrassv2 describe-component \
  --arn "arn:aws:greengrass:region:account-
id:components:com.example.HelloWorldLambda:versions:1.0.0"
```

If the component validates, the response indicates that the component state is `DEPLOYABLE`.

```
{
  "arn": "arn:aws:greengrass:region:account-
id:components:com.example.HelloWorldLambda:versions:1.0.0",
  "componentName": "com.example.HelloWorldLambda",
  "componentVersion": "1.0.0",
  "creationTimestamp": "2020-12-15T20:56:34.376000-08:00",
  "publisher": "AWS Lambda",
  "status": {
    "componentState": "DEPLOYABLE",
    "message": "NONE",
    "errors": {}
  },
  "platforms": [
    {
      "name": "Linux x86",
      "attributes": {
        "architecture": "x86",
        "os": "linux"
      }
    }
  ]
}
```

After the component is `DEPLOYABLE`, you can deploy the Lambda function to your core devices. For more information, see Deploy AWS IoT Greengrass components to devices.

Use the AWS IoT Device SDK to communicate with the Greengrass nucleus, other components, and AWS IoT Core

Components running on your core device can use the AWS IoT Greengrass Core interprocess communication (IPC) library in the AWS IoT Device SDK to communicate with the AWS IoT Greengrass nucleus and other Greengrass components. To develop and run custom components that use IPC, you must use the AWS IoT Device SDK to connect to the AWS IoT Greengrass Core IPC service and perform IPC operations.

The IPC interface supports two types of operations:

- **Request/response**

  Components send a request to the IPC service and receive a response that contains the result of the request.
- **Subscription**

  Components send a subscription request to the IPC service and expect a stream of event messages in response. Components provide a subscription handler that handles event messages, errors, and stream closure. The AWS IoT Device SDK includes a handler interface with the correct response and event types for each IPC operation. For more information, see Subscribe to IPC event streams.

Topics

- IPC client versions
- Supported SDKs for interprocess communication
- Connect to the AWS IoT Greengrass Core IPC service
- Authorize components to perform IPC operations
- Subscribe to IPC event streams
- IPC best practices
- Publish/subscribe local messages
- Publish/subscribe AWS IoT Core MQTT messages
- Interact with component lifecycle

- Interact with component configuration
- Retrieve secret values
- Interact with local shadows
- Manage local deployments and components
- Authenticate and authorize client devices

IPC client versions

In later versions of the Java and Python SDKs, AWS IoT Greengrass provides an improved version of the IPC client, called IPC client V2. IPC client V2:

- Reduces the amount of code that you need to write to use IPC operations and helps avoid common errors that can occur with IPC client V1.
- Calls subscription handler callbacks in a separate thread, so you can now run blocking code, including additional IPC function calls, in subscription handler callbacks. IPC client V1 uses the same thread to communicate with the IPC server and call subscription handler callbacks.
- Lets you call subscription operations using Lambda expressions (Java) or functions (Python). IPC client V1 requires you to define subscription handler classes.
- Provides synchronous and asynchronous versions of each IPC operation. IPC client V1 provides only asynchronous versions of each operation.

We recommend that you use IPC client V2 to take advantage of these improvements. However, many examples in this documentation and in some online content demonstrate only how to use IPC client V1. You can use the following examples and tutorials to see sample components that use IPC client V2:

- PublishToTopic examples
- SubscribeToTopic examples
- Tutorial: Develop a Greengrass component that defers component updates
- Tutorial: Interact with local IoT devices over MQTT

Currently, the AWS IoT Device SDK for C++ v2 supports only IPC client V1.

For components written in C, C++, or Rust, the AWS IoT Greengrass Component SDK provides lightweight bindings for a subset of Greengrass IPC operations. Its small footprint also makes it

suitable for resource-constrained devices. You can also use the AWS IoT Device SDK for C++ v2, which supports IPC client V1. For more information, see Use AWS IoT Greengrass Component SDK (C, C++, Rust).

Supported SDKs for interprocess communication

The AWS IoT Greengrass Core IPC libraries are included in the following SDKs.

SDK	Minimum version	Usage
AWS IoT Greengrass Component SDK (C, C++, Rust)	v1.0.0	See Use AWS IoT Greengrass Component SDK (C, C++, Rust)
AWS IoT Device SDK for Java v2	v1.6.0	See Use AWS IoT Device SDK for Java v2 (IPC client V2)
AWS IoT Device SDK for Python v2	v1.9.0	See Use AWS IoT Device SDK for Python v2 (IPC client V2)
AWS IoT Device SDK for C++ v2	v1.17.0	See Use AWS IoT Device SDK for C++ v2
AWS IoT Device SDK for JavaScript v2	v1.12.0	See Use AWS IoT Device SDK for JavaScript v2 (IPC client V1)

Connect to the AWS IoT Greengrass Core IPC service

To use interprocess communication in your custom component, you must create a connection to an IPC server socket that the AWS IoT Greengrass Core software runs. Complete the following tasks

to download and use the AWS IoT Device SDK or the AWS IoT Greengrass Component SDK in the language of your choice.

Use AWS IoT Greengrass Component SDK (C, C++, Rust)

The AWS IoT Greengrass Component SDK provides C, C++, and Rust bindings for Greengrass IPC operations. Its small footprint also makes it suitable for resource-constrained devices.

To use the AWS IoT Greengrass Component SDK

1. Download or clone the AWS IoT Greengrass Component SDK (v1.0.0 or later).
2. Follow the installation instructions in the README for your language (C, C++, or Rust).
3. Use the IPC client in your component code. For examples, see the IPC operation pages in this guide, which include AWS IoT Greengrass Component SDK code examples in the C, C++ (Component SDK), and Rust tabs.

Use AWS IoT Device SDK for Java v2 (IPC client V2)

To use the AWS IoT Device SDK for Java v2 (IPC client V2)

1. Download the AWS IoT Device SDK for Java v2 (v1.6.0 or later).
2. Do one of the following to run your custom code in your component:
   - Build your component as a JAR file that includes the AWS IoT Device SDK, and run this JAR file in your component recipe.
   - Define the AWS IoT Device SDK JAR as a component artifact, and add that artifact to the classpath when you run your application in your component recipe.
3. Use the following code to create the IPC client.

```
try (GreengrassCoreIPCClientV2 ipcClient =
 GreengrassCoreIPCClientV2.builder().build()) {
    // Use client.
} catch (Exception e) {
    LOGGER.log(Level.SEVERE, "Exception occurred when using IPC.", e);
    System.exit(1);
}
```

Use AWS IoT Device SDK for Python v2 (IPC client V2)

To use the AWS IoT Device SDK for Python v2 (IPC client V2)

1. Download the AWS IoT Device SDK for Python (v1.9.0 or later).
2. Add the SDK's installation steps to the install lifecycle in your component's recipe.
3. Create a connection to the AWS IoT Greengrass Core IPC service. Use the following code to create the IPC client.

```
from awsiot.greengrasscoreipc.clientv2 import GreengrassCoreIPCClientV2

try:
    ipc_client = GreengrassCoreIPCClientV2()
    # Use IPC client.
except Exception:
    print('Exception occurred when using IPC.', file=sys.stderr)
    traceback.print_exc()
    exit(1)
```

Use AWS IoT Device SDK for C++ v2

To build the AWS IoT Device SDK v2 for C++, a device must have the following tools:

- C++ 11 or later
- CMake 3.1 or later
- One of the following compilers:
  - GCC 4.8 or later
  - Clang 3.9 or later
  - MSVC 2015 or later

To use the AWS IoT Device SDK for C++ v2

1. Download the AWS IoT Device SDK for C++ v2 (v1.17.0 or later).
2. Follow the installation instructions in the README to build the AWS IoT Device SDK for C++ v2 from source.

3. In your C++ build tool, link the Greengrass IPC library, `AWS::GreengrassIpc-cpp`, that you built in the previous step. The following `CMakeLists.txt` example links the Greengrass IPC library to a project that you build with CMake.

```
cmake_minimum_required(VERSION 3.1)
project (greengrassv2_pubsub_subscriber)

file(GLOB MAIN_SRC
        "*.h"
        "*.cpp"
        )
add_executable(${PROJECT_NAME} ${MAIN_SRC})

set_target_properties(${PROJECT_NAME} PROPERTIES
        LINKER_LANGUAGE CXX
        CXX_STANDARD 11)
find_package(aws-crt-cpp PATHS ~/sdk-cpp-workspace/build)
find_package(EventstreamRpc-cpp PATHS ~/sdk-cpp-workspace/build)
find_package(GreengrassIpc-cpp PATHS ~/sdk-cpp-workspace/build)
target_link_libraries(${PROJECT_NAME} AWS::GreengrassIpc-cpp)
```

4. In your component code, create a connection to the AWS IoT Greengrass Core IPC service to create an IPC client (`Aws::Greengrass::GreengrassCoreIpcClient`). You must define an IPC connection lifecycle handler that handles IPC connection, disconnection, and error events. The following example creates an IPC client and an IPC connection lifecycle handler that prints when the IPC client connects, disconnects, and encounters errors.

```
#include <iostream>

#include <aws/crt/Api.h>
#include <aws/greengrass/GreengrassCoreIpcClient.h>

using namespace Aws::Crt;
using namespace Aws::Greengrass;

class IpcClientLifecycleHandler : public ConnectionLifecycleHandler {
    void OnConnectCallback() override {
        std::cout << "OnConnectCallback" << std::endl;
    }

    void OnDisconnectCallback(RpcError error) override {
```

```
        std::cout << "OnDisconnectCallback: " << error.StatusToString() <<
 std::endl;
        exit(-1);
    }

    bool OnErrorCallback(RpcError error) override {
        std::cout << "OnErrorCallback: " << error.StatusToString() << std::endl;
        return true;
    }
};

int main() {
    // Create the IPC client.
    ApiHandle apiHandle(g_allocator);
    Io::EventLoopGroup eventLoopGroup(1);
    Io::DefaultHostResolver socketResolver(eventLoopGroup, 64, 30);
    Io::ClientBootstrap bootstrap(eventLoopGroup, socketResolver);
    IpcClientLifecycleHandler ipcLifecycleHandler;
    GreengrassCoreIpcClient ipcClient(bootstrap);
    auto connectionStatus = ipcClient.Connect(ipcLifecycleHandler).get();
    if (!connectionStatus) {
        std::cerr << "Failed to establish IPC connection: " <<
 connectionStatus.StatusToString() << std::endl;
        exit(-1);
    }

    // Use the IPC client to create an operation request.

    // Activate the operation request.
    auto activate = operation.Activate(request, nullptr);
    activate.wait();

    // Wait for Greengrass Core to respond to the request.
    auto responseFuture = operation.GetResult();
    if (responseFuture.wait_for(std::chrono::seconds(timeout)) ==
 std::future_status::timeout) {
        std::cerr << "Operation timed out while waiting for response from
 Greengrass Core." << std::endl;
        exit(-1);
    }

    // Check the result of the request.
    auto response = responseFuture.get();
    if (response) {
```

```
        std::cout << "Successfully published to topic: " << topic << std::endl;
    } else {
        // An error occurred.
        std::cout << "Failed to publish to topic: " << topic << std::endl;
        auto errorType = response.GetResultType();
        if (errorType == OPERATION_ERROR) {
            auto *error = response.GetOperationError();
            std::cout << "Operation error: " << error->GetMessage().value() <<
 std::endl;
        } else {
            std::cout << "RPC error: " << response.GetRpcError() << std::endl;
        }
        exit(-1);
    }

    return 0;
}
```

5. To run your custom code in your component, build your code as a binary artifact, and run the binary artifact in your component recipe. Set the artifact's `Execute` permission to `OWNER` to enable the AWS IoT Greengrass Core software to run the binary artifact.

   Your component recipe's `Manifests` section might look similar to the following example.

   JSON

   ```
   {
     ...
     "Manifests": [
       {
         "Lifecycle": {
           "Run": "{artifacts:path}/greengrassv2_pubsub_subscriber"
         },
         "Artifacts": [
           {
             "URI": "s3://amzn-s3-demo-bucket/artifacts/
   com.example.PubSubSubscriberCpp/1.0.0/greengrassv2_pubsub_subscriber",
             "Permission": {
               "Execute": "OWNER"
             }
           }
         ]
       }
   ```

```
  ]
}
```

YAML

```
...
Manifests:
  - Lifecycle:
      Run: {artifacts:path}/greengrassv2_pubsub_subscriber
    Artifacts:
      - URI: s3://amzn-s3-demo-bucket/artifacts/
com.example.PubSubSubscriberCpp/1.0.0/greengrassv2_pubsub_subscriber
        Permission:
          Execute: OWNER
```

Use AWS IoT Device SDK for JavaScript v2 (IPC client V1)

To build the AWS IoT Device SDK for JavaScript v2 for use with NodeJS, a device must have the following tools:

- NodeJS 10.0 or later
  - Run `node -v` to check the Node version.
- CMake 3.1 or later

To use the AWS IoT Device SDK for JavaScript v2 (IPC client V1)

1. Download the AWS IoT Device SDK for JavaScript v2 (v1.12.10 or later).
2. Follow the installation instructions in the README to build the AWS IoT Device SDK for JavaScript v2 from source.
3. Create a connection to the AWS IoT Greengrass Core IPC service. Complete the following steps to create the IPC client and establish a connection.
4. Use the following code to create the IPC client.

```
import * as greengrascoreipc from 'aws-iot-device-sdk-v2';

let client = greengrascoreipc.createClient();
```

5. Use the following code to establish a connection from your component to the Greengrass nucleus.

```
await client.connect();
```

Authorize components to perform IPC operations

To allow your custom components to use some IPC operations, you must define *authorization policies* that allow the component to perform the operation on certain resources. Each authorization policy defines a list of operations and a list of resources that the policy allows. For example, the publish/subscribe messaging IPC service defines publish and subscribe operations for topic resources. You can use the * wildcard to allow access to all operations or all resources.

You define authorization policies with the `accessControl` configuration parameter, which you can set in the component recipe or when you deploy the component. The `accessControl` object maps IPC service identifiers to lists of authorization policies. You can define multiple authorization policies for each IPC service to control access. Each authorization policy has a policy ID, which must be unique among all components.

Tip

To create unique policy IDs, you can combine the component name, IPC service name, and a counter. For example, a component named `com.example.HelloWorld` might define two publish/subscribe authorization policies with the following IDs:

- `com.example.HelloWorld:pubsub:1`
- `com.example.HelloWorld:pubsub:2`

Authorization policies use the following format. This object is the `accessControl` configuration parameter.

JSON

```
{
  "IPC service identifier": {
    "policyId": {
      "policyDescription": "description",
```

```
      "operations": [
        "operation1",
        "operation2"
      ],
      "resources": [
        "resource1",
        "resource2"
      ]
    }
  }
}
```

YAML

```
IPC service identifier:
  policyId:
    policyDescription: description
    operations:
      - operation1
      - operation2
    resources:
      - resource1
      - resource2
```

Wildcards in authorization policies

You can use the * wildcard in the `resources` element of IPC authorization policies to allow access to multiple resources in a single authorization policy.

- In all versions of the Greengrass nucleus, you can specify a single * character as a resource to allow access to all resources.
- In Greengrass nucleus v2.6.0 and later, you can specify the * character in a resource to match any combination of characters. For example, you can specify `factory/1/devices/Thermostat*/status` to allow access to a status topic for all thermostat devices in a factory, where each device's name begins with `Thermostat`.

When you define authorization policies for the AWS IoT Core MQTT IPC service, you can also use MQTT wildcards (+ and #) to match multiple resources. For more information, see MQTT wildcards in AWS IoT Core MQTT IPC authorization policies.

Recipe variables in authorization policies

If you use Greengrass nucleus v2.6.0 or later, and you set the Greengrass nucleus' interpolateComponentConfiguration configuration option to `true`, you can use the `{iot:thingName}` recipe variable in authorization policies. When you need an authorization policy that includes the core device's name, such as for MQTT topics or device shadows, you can use this recipe variable to configure a single authorization policy for a group of core devices. For example, you can allow a component access to the following resource for shadow IPC operations.

```
$aws/things/{iot:thingName}/shadow/
```

Special characters in authorization policies

To specify a literal * or ? character in an authorization policy, you must use an escape sequence. The following escape sequences instruct the AWS IoT Greengrass Core software to use the literal value instead of the character's special meaning. For example, the * character is a wildcard that matches any combination of characters.

Literal character	Escape sequence	Notes
*	${*}	
?	${?}	AWS IoT Greengrass doesn't currently support the ? wildcard, which matches any single character.
$	${$}	Use this escape sequence to match a resource that contains ${. For example, to match a resource named ${resourceName} , you must specify ${${}{resourceName} . Otherwise , to match a resource that contains $, you can use a literal $, such as to allow

Literal character	Escape sequence	Notes
		access to a topic that begins with $aws.

Authorization policy examples

You can reference the following authorization policy examples to help you configure authorization policies for your components.

Example component recipe with an authorization policy

The following example component recipe includes an `accessControl` object defines an authorization policy. This policy authorizes the `com.example.HelloWorld` component to publish to the `test/topic` topic.

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.HelloWorld",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that publishes messages.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
          "com.example.HelloWorld:pubsub:1": {
            "policyDescription": "Allows access to publish to test/topic.",
            "operations": [
              "aws.greengrass#PublishToTopic"
            ],
            "resources": [
              "test/topic"
            ]
          }
        }
      }
    }
  },
  "Manifests": [
```

```
    {
      "Lifecycle": {
        "Run": "java -jar {artifacts:path}/HelloWorld.jar"
      }
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.HelloWorld
ComponentVersion: '1.0.0'
ComponentDescription: A component that publishes messages.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    accessControl:
      aws.greengrass.ipc.pubsub:
        "com.example.HelloWorld:pubsub:1":
          policyDescription: Allows access to publish to test/topic.
          operations:
            - "aws.greengrass#PublishToTopic"
          resources:
            - "test/topic"
Manifests:
  - Lifecycle:
      Run: |-
        java -jar {artifacts:path}/HelloWorld.jar
```

Example component configuration update with an authorization policy

The following example configuration update in a deployment specifies to configure a component with an `accessControl` object that defines an authorization policy. This policy authorizes the `com.example.HelloWorld` component to publish to the `test/topic` topic.

Console

Configuration to merge

```
{
```

```
  "accessControl": {
    "aws.greengrass.ipc.pubsub": {
      "com.example.HelloWorld:pubsub:1": {
        "policyDescription": "Allows access to publish to test/topic.",
        "operations": [
          "aws.greengrass#PublishToTopic"
        ],
        "resources": [
          "test/topic"
        ]
      }
    }
  }
}
```

AWS CLI

The following command creates a deployment to a core device.

```
aws greengrassv2 create-deployment --cli-input-json file://hello-world-
deployment.json
```

The `hello-world-deployment.json` file contains the following JSON document.

```
{
  "targetArn": "arn:aws:iot:us-west-2:123456789012:thing/MyGreengrassCore",
  "deploymentName": "Deployment for MyGreengrassCore",
  "components": {
    "com.example.HelloWorld": {
      "componentVersion": "1.0.0",
      "configurationUpdate": {
        "merge": "{\"accessControl\":{\"aws.greengrass.ipc.pubsub\":
{\"com.example.HelloWorld:pubsub:1\":{\"policyDescription\":\"Allows access to
 publish to test/topic.\",\"operations\":[\"aws.greengrass#PublishToTopic\"],
\"resources\":[\"test/topic\"]}}}}"
      }
    }
  }
}
```

Greengrass CLI

The following Greengrass CLI command creates a local deployment on a core device.

```
sudo greengrass-cli deployment create \
  --recipeDir recipes \
  --artifactDir artifacts \
  --merge "com.example.HelloWorld=1.0.0" \
  --update-config hello-world-configuration.json
```

The `hello-world-configuration.json` file contains the following JSON document.

```
{
  "com.example.HelloWorld": {
    "MERGE": {
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
          "com.example.HelloWorld:pubsub:1": {
            "policyDescription": "Allows access to publish to test/topic.",
            "operations": [
              "aws.greengrass#PublishToTopic"
            ],
            "resources": [
              "test/topic"
            ]
          }
        }
      }
    }
  }
}
```

Subscribe to IPC event streams

You can use IPC operations to subscribe to streams of events on a Greengrass core device. To use a subscribe operation, define a *subscription handler* and create a request to the IPC service. Then, the IPC client runs the subscription handler's functions each time that the core device streams an event message to your component.

You can close a subscription to stop processing event messages. To do so, call `closeStream()` (Java), `close()` (Python), or `Close()` (C++) on the subscription operation object that you used to open the subscription.

The AWS IoT Greengrass Core IPC service supports the following subscribe operations:

- SubscribeToTopic
- SubscribeToIoTCore
- SubscribeToComponentUpdates
- SubscribeToConfigurationUpdate
- SubscribeToValidateConfigurationUpdates

Topics

- Define subscription handlers
- Example subscription handlers

Define subscription handlers

To define a subscription handler, define callback functions that handle event messages, errors, and stream closure. If you use IPC client V1, you must define these functions in a class. If you use IPC client V2, which is available in later versions of the Java and Python SDKs, you can define these functions without creating a subscription handler class.

Java

If you use IPC client V1, you must implement the generic `software.amazon.awssdk.eventstreamrpc.StreamResponseHandler<`*`StreamEventType`*`>` interface. *`StreamEventType`* is the type of event message for the subscription operation. Define the following functions to handle event messages, errors, and stream closure.

If you use IPC client V2, you can define these functions outside of a subscription handler class or use lambda expressions.

```
void onStreamEvent(StreamEventType event)
```

The callback that the IPC client calls when it receives an event message, such as an MQTT message or a component update notification.

```
boolean onStreamError(Throwable error)
```

The callback that the IPC client calls when a stream error occurs.

Return true to close the subscription stream as a result of the error, or return false to keep the stream open.

```
void onStreamClosed()
```

The callback that the IPC client calls when the stream closes.

Python

If you use IPC client V1, you must extend the stream response handler class that corresponds to the subscription operation. The AWS IoT Device SDK includes a subscription handler class for each subscription operation. *`StreamEventType`* is the type of event message for the subscription operation. Define the following functions to handle event messages, errors, and stream closure.

If you use IPC client V2, you can define these functions outside of a subscription handler class or use lambda expressions.

```
def on_stream_event(self, event: StreamEventType) -> None
```

The callback that the IPC client calls when it receives an event message, such as an MQTT message or a component update notification.

```
def on_stream_error(self, error: Exception) -> bool
```

The callback that the IPC client calls when a stream error occurs.

Return true to close the subscription stream as a result of the error, or return false to keep the stream open.

```
def on_stream_closed(self) -> None
```

The callback that the IPC client calls when the stream closes.

C++ (IPC client V1)

Implement a class that derives from the stream response handler class that corresponds to the subscription operation. The AWS IoT Device SDK includes a subscription handler base class

for each subscription operation. *StreamEventType* is the type of event message for the subscription operation. Define the following functions to handle event messages, errors, and stream closure.

`void OnStreamEvent(StreamEventType *event)`

The callback that the IPC client calls when it receives an event message, such as an MQTT message or a component update notification.

`bool OnStreamError(OperationError *error)`

The callback that the IPC client calls when a stream error occurs.

Return true to close the subscription stream as a result of the error, or return false to keep the stream open.

`void OnStreamClosed()`

The callback that the IPC client calls when the stream closes.

JavaScript

Implement a class that derives from the stream response handler class that corresponds to the subscription operation. The AWS IoT Device SDK includes a subscription handler base class for each subscription operation. *StreamEventType* is the type of event message for the subscription operation. Define the following functions to handle event messages, errors, and stream closure.

`on(event: 'ended', listener: StreamingOperationEndedListener)`

The callback that the IPC client calls when the stream closes.

`on(event: 'streamError', listener: StreamingRpcErrorListener)`

The callback that the IPC client calls when a stream error occurs.

Return true to close the subscription stream as a result of the error, or return false to keep the stream open.

`on(event: 'message', listener: (message: InboundMessageType) => void)`

The callback that the IPC client calls when it receives an event message, such as an MQTT message or a component update notification.

Example subscription handlers

The following example demonstrates how to use the SubscribeToTopic operation and a subscription handler to subscribe to local publish/subscribe messages.

Java (IPC client V2)

Example: Subscribe to local publish/subscribe messages

```
package com.aws.greengrass.docs.samples.ipc;

import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClientV2;
import software.amazon.awssdk.aws.greengrass.SubscribeToTopicResponseHandler;
import software.amazon.awssdk.aws.greengrass.model.*;

import java.nio.charset.StandardCharsets;
import java.util.Optional;

public class SubscribeToTopicV2 {

    public static void main(String[] args) {
        String topic = args[0];
        try (GreengrassCoreIPCClientV2 ipcClient =
 GreengrassCoreIPCClientV2.builder().build()) {
            SubscribeToTopicRequest request = new
 SubscribeToTopicRequest().withTopic(topic);
            GreengrassCoreIPCClientV2.StreamingResponse<SubscribeToTopicResponse,
                    SubscribeToTopicResponseHandler> response =
                    ipcClient.subscribeToTopic(request,
 SubscribeToTopicV2::onStreamEvent,
                            Optional.of(SubscribeToTopicV2::onStreamError),
                            Optional.of(SubscribeToTopicV2::onStreamClosed));
            SubscribeToTopicResponseHandler responseHandler =
 response.getHandler();
            System.out.println("Successfully subscribed to topic: " + topic);

            // Keep the main thread alive, or the process will exit.
            try {
                while (true) {
                    Thread.sleep(10000);
                }
            } catch (InterruptedException e) {
                System.out.println("Subscribe interrupted.");
```

```
            }

            // To stop subscribing, close the stream.
            responseHandler.closeStream();
        } catch (Exception e) {
            if (e.getCause() instanceof UnauthorizedError) {
                System.err.println("Unauthorized error while publishing to topic: "
+ topic);
            } else {
                System.err.println("Exception occurred when using IPC.");
            }
            e.printStackTrace();
            System.exit(1);
        }
    }

    public static void onStreamEvent(SubscriptionResponseMessage
subscriptionResponseMessage) {
        try {
            BinaryMessage binaryMessage =
subscriptionResponseMessage.getBinaryMessage();
            String message = new String(binaryMessage.getMessage(),
StandardCharsets.UTF_8);
            String topic = binaryMessage.getContext().getTopic();
            System.out.printf("Received new message on topic %s: %s%n", topic,
message);
        } catch (Exception e) {
            System.err.println("Exception occurred while processing subscription
response " +
                    "message.");
            e.printStackTrace();
        }
    }

    public static boolean onStreamError(Throwable error) {
        System.err.println("Received a stream error.");
        error.printStackTrace();
        return false; // Return true to close stream, false to keep stream open.
    }

    public static void onStreamClosed() {
        System.out.println("Subscribe to topic stream closed.");
    }
```

```
}
```

Python (IPC client V2)

Example: Subscribe to local publish/subscribe messages

```
import sys
import time
import traceback

from awsiot.greengrasscoreipc.clientv2 import GreengrassCoreIPCClientV2
from awsiot.greengrasscoreipc.model import (
    SubscriptionResponseMessage,
    UnauthorizedError
)


def main():
    args = sys.argv[1:]
    topic = args[0]

    try:
        ipc_client = GreengrassCoreIPCClientV2()
        # Subscription operations return a tuple with the response and the
 operation.
        _, operation = ipc_client.subscribe_to_topic(topic=topic,
 on_stream_event=on_stream_event,

 on_stream_error=on_stream_error, on_stream_closed=on_stream_closed)
        print('Successfully subscribed to topic: ' + topic)

        # Keep the main thread alive, or the process will exit.
        try:
            while True:
                time.sleep(10)
        except InterruptedError:
            print('Subscribe interrupted.')

        # To stop subscribing, close the stream.
        operation.close()
    except UnauthorizedError:
        print('Unauthorized error while subscribing to topic: ' +
              topic, file=sys.stderr)
        traceback.print_exc()
```

```
        exit(1)
    except Exception:
        print('Exception occurred', file=sys.stderr)
        traceback.print_exc()
        exit(1)


def on_stream_event(event: SubscriptionResponseMessage) -> None:
    try:
        message = str(event.binary_message.message, 'utf-8')
        topic = event.binary_message.context.topic
        print('Received new message on topic %s: %s' % (topic, message))
    except:
        traceback.print_exc()


def on_stream_error(error: Exception) -> bool:
    print('Received a stream error.', file=sys.stderr)
    traceback.print_exc()
    return False  # Return True to close stream, False to keep stream open.


def on_stream_closed() -> None:
    print('Subscribe to topic stream closed.')


if __name__ == '__main__':
    main()
```

C++ (IPC client V1)

Example: Subscribe to local publish/subscribe messages

```
#include <iostream>

#include </crt/Api.h>
#include <aws/greengrass/GreengrassCoreIpcClient.h>

using namespace Aws::Crt;
using namespace Aws::Greengrass;

class SubscribeResponseHandler : public SubscribeToTopicStreamHandler {
    public:
        virtual ~SubscribeResponseHandler() {}
```

```
    private:
        void OnStreamEvent(SubscriptionResponseMessage *response) override {
            auto jsonMessage = response->GetJsonMessage();
            if (jsonMessage.has_value() &&
 jsonMessage.value().GetMessage().has_value()) {
                auto messageString =
 jsonMessage.value().GetMessage().value().View().WriteReadable();
                // Handle JSON message.
            } else {
                auto binaryMessage = response->GetBinaryMessage();
                if (binaryMessage.has_value() &&
 binaryMessage.value().GetMessage().has_value()) {
                    auto messageBytes = binaryMessage.value().GetMessage().value();
                    std::string messageString(messageBytes.begin(),
 messageBytes.end());
                    // Handle binary message.
                }
            }
        }

        bool OnStreamError(OperationError *error) override {
            // Handle error.
            return false; // Return true to close stream, false to keep stream open.
        }

        void OnStreamClosed() override {
            // Handle close.
        }
};

class IpcClientLifecycleHandler : public ConnectionLifecycleHandler {
    void OnConnectCallback() override {
        // Handle connection to IPC service.
    }

    void OnDisconnectCallback(RpcError error) override {
        // Handle disconnection from IPC service.
    }

    bool OnErrorCallback(RpcError error) override {
        // Handle IPC service connection error.
        return true;
    }
```

```
};

int main() {
    ApiHandle apiHandle(g_allocator);
    Io::EventLoopGroup eventLoopGroup(1);
    Io::DefaultHostResolver socketResolver(eventLoopGroup, 64, 30);
    Io::ClientBootstrap bootstrap(eventLoopGroup, socketResolver);
    IpcClientLifecycleHandler ipcLifecycleHandler;
    GreengrassCoreIpcClient ipcClient(bootstrap);
    auto connectionStatus = ipcClient.Connect(ipcLifecycleHandler).get();
    if (!connectionStatus) {
        std::cerr << "Failed to establish IPC connection: " <<
 connectionStatus.StatusToString() << std::endl;
        exit(-1);
    }

    String topic("my/topic");
    int timeout = 10;

    SubscribeToTopicRequest request;
    request.SetTopic(topic);

    //SubscribeResponseHandler streamHandler;
    auto streamHandler = MakeShared<SubscribeResponseHandler>(DefaultAllocator());
    auto operation = ipcClient.NewSubscribeToTopic(streamHandler);
    auto activate = operation->Activate(request, nullptr);
    activate.wait();

    auto responseFuture = operation->GetResult();
    if (responseFuture.wait_for(std::chrono::seconds(timeout)) ==
 std::future_status::timeout) {
        std::cerr << "Operation timed out while waiting for response from Greengrass
 Core." << std::endl;
        exit(-1);
    }

    auto response = responseFuture.get();
    if (!response) {
        // Handle error.
        auto errorType = response.GetResultType();
        if (errorType == OPERATION_ERROR) {
            auto *error = response.GetOperationError();
            (void)error;
            // Handle operation error.
```

```
        } else {
            // Handle RPC error.
        }
        exit(-1);
    }

    // Keep the main thread alive, or the process will exit.
    while (true) {
        std::this_thread::sleep_for(std::chrono::seconds(10));
    }

    operation->Close();
    return 0;
}
```

JavaScript

Example: Subscribe to local publish/subscribe messages

```
import * as greengrasscoreipc from "aws-iot-device-sdk-v2/dist/greengrasscoreipc";
import {SubscribeToTopicRequest, SubscriptionResponseMessage} from "aws-iot-device-
sdk-v2/dist/greengrasscoreipc/model";
import {RpcError} from "aws-iot-device-sdk-v2/dist/eventstream_rpc";

class SubscribeToTopic {
    private ipcClient : greengrasscoreipc.Client
    private readonly topic : string;

    constructor() {
        // define your own constructor, e.g.
        this.topic = "<define_your_topic>";
        this.subscribeToTopic().then(r => console.log("Started workflow"));
    }

    private async subscribeToTopic() {
        try {
            this.ipcClient = await getIpcClient();

            const subscribeToTopicRequest : SubscribeToTopicRequest = {
                topic: this.topic,
            }
```

```
            const streamingOperation =
 this.ipcClient.subscribeToTopic(subscribeToTopicRequest, undefined); //
 conditionally apply options

            streamingOperation.on("message", (message: SubscriptionResponseMessage)
 => {
                // parse the message depending on your use cases, e.g.
                if(message.binaryMessage && message.binaryMessage.message) {
                    const receivedMessage =
 message.binaryMessage?.message.toString();
                }
            });

            streamingOperation.on("streamError", (error : RpcError) => {
                // define your own error handling logic
            })

            streamingOperation.on("ended", () => {
                // define your own logic
            })

            await streamingOperation.activate();

            // Keep the main thread alive, or the process will exit.
            await new Promise((resolve) => setTimeout(resolve, 10000))
        } catch (e) {
            // parse the error depending on your use cases
            throw e
        }
    }
}

export async function getIpcClient(){
    try {
        const ipcClient = greengrasscoreipc.createClient();
        await ipcClient.connect()
            .catch(error => {
                // parse the error depending on your use cases
                throw error;
            });
        return ipcClient
    } catch (err) {
        // parse the error depending on your use cases
        throw err
```

```
    }
}

// starting point
const subscribeToTopic = new SubscribeToTopic();
```

Rust

Example: Subscribe to local publish/subscribe messages

```
use gg_sdk::{Sdk, SubscribeToTopicPayload};
use std::{thread, time::Duration};

fn main() {
    let sdk = Sdk::init();
    sdk.connect().expect("Failed to establish IPC connection");

    let topic = "my/topic";

    let callback = |topic: &str, payload: SubscribeToTopicPayload| match payload
    {
        SubscribeToTopicPayload::Binary(message) => {
            let message = String::from_utf8_lossy(message);
            println!("Received new message on topic {topic}: {message}");
        }
        SubscribeToTopicPayload::Json(_) => {
            println!("Received new message on topic {topic}: (JSON message)");
        }
    };

    let _sub = sdk
        .subscribe_to_topic(topic, &callback)
        .expect("Failed to subscribe to topic");

    println!("Successfully subscribed to topic: {topic}");

    // Keep the main thread alive, or the process will exit.
    loop {
        thread::sleep(Duration::from_secs(10));
    }
}
```

C

Example: Subscribe to local publish/subscribe messages

```
#include <assert.h>
#include <gg/error.h>
#include <gg/ipc/client.h>
#include <gg/object.h>
#include <gg/sdk.h>
#include <gg/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

static void on_subscription_response(
    void *ctx, GgBuffer topic, GgObject payload, GgIpcSubscriptionHandle handle
) {
    (void) ctx;
    (void) handle;

    if (gg_obj_type(payload) == GG_TYPE_BUF) {
        GgBuffer message = gg_obj_into_buf(payload);
        printf(
            "Received new message on topic %.*s: %.*s\n",
            (int) topic.len,
            topic.data,
            (int) message.len,
            message.data
        );
    } else {
        assert(gg_obj_type(payload) == GG_TYPE_MAP);
        printf(
            "Received new message on topic %.*s: (JSON message)\n",
            (int) topic.len,
            topic.data
        );
    }
}

int main(void) {
    gg_sdk_init();

    GgError err = ggipc_connect();
    if (err != GG_ERR_OK) {
```

```
        fprintf(stderr, "Failed to establish IPC connection.\n");
        exit(-1);
    }

    GgBuffer topic = GG_STR("my/topic");

    GgIpcSubscriptionHandle handle;
    err = ggipc_subscribe_to_topic(
        topic, on_subscription_response, NULL, &handle
    );
    if (err != GG_ERR_OK) {
        fprintf(
            stderr,
            "Failed to subscribe to topic: %.*s\n",
            (int) topic.len,
            topic.data
        );
        exit(-1);
    }

    printf(
        "Successfully subscribed to topic: %.*s\n", (int) topic.len, topic.data
    );

    // Keep the main thread alive, or the process will exit.
    while (1) {
        sleep(10);
    }

    // To stop subscribing, close the stream.
    ggipc_close_subscription(handle);
}
```

C++ (Component SDK)

Example: Subscribe to local publish/subscribe messages

```
#include <gg/ipc/client.hpp>
#include <gg/object.hpp>
#include <unistd.h>
#include <cassert>
#include <iostream>

class ResponseHandler : public gg::ipc::LocalTopicCallback {
```

```
    void operator()(
        std::string_view topic,
        gg::Object payload,
        gg::ipc::Subscription &handle
    ) override {
        (void) handle;
        if (payload.index() == GG_TYPE_BUF) {
            std::cout << "Received new message on topic " << topic << ": "
                      << get<gg::Buffer>(payload) << "\n";
        } else {
            assert(payload.index() == GG_TYPE_MAP);
            std::cout << "Received new message on topic " << topic
                      << ": (JSON message)\n";
        }
    }
};

int main() {
    auto &client = gg::ipc::Client::get();

    auto error = client.connect();
    if (error) {
        std::cerr << "Failed to establish IPC connection.\n";
        exit(-1);
    }

    std::string_view topic = "my/topic";

    static ResponseHandler handler;
    error = client.subscribe_to_topic(topic, handler);
    if (error) {
        std::cerr << "Failed to subscribe to topic: " << topic << "\n";
        exit(-1);
    }

    std::cout << "Successfully subscribed to topic: " << topic << "\n";

    // Keep the main thread alive, or the process will exit.
    while (1) {
        sleep(10);
    }
}
```

IPC best practices

The best practices for using IPC in custom components differ between IPC client V1 and IPC client V2. Follow the best practices for the IPC client version that you use.

IPC client V2

The IPC client V2 runs callback functions in a separate thread, so compared to IPC client V1, there are fewer guidelines for you to follow when you use IPC and write subscription handler functions.

- **Reuse one IPC client**

  After you create an IPC client, keep it open and reuse it for all IPC operations. Creating multiple clients uses extra resources and can result in resource leaks.

- **Handle exceptions**

  The IPC client V2 logs uncaught exceptions in subscription handler functions. You should catch exceptions in your handler functions to handle errors that occur in your code.

IPC client V1

The IPC client V1 uses a single thread that communicates with the IPC server and calls subscription handlers. You must consider this synchronous behavior when you write subscription handler functions.

- **Reuse one IPC client**

  After you create an IPC client, keep it open and reuse it for all IPC operations. Creating multiple clients uses extra resources and can result in resource leaks.

- **Run blocking code asynchronously**

  The IPC client V1 can't send new requests or process new event messages while the thread is blocked. You should run blocking code in a separate thread that you run from the handler function. Blocking code includes `sleep` calls, loops that continuously run, and synchronous I/O requests that take time to complete.

- **Send new IPC requests asynchronously**

The IPC client V1 can't send a new request from within subscription handler functions, because the request blocks the handler function if you wait for a response. You should send IPC requests in a separate thread that you run from the handler function.

- **Handle exceptions**

  The IPC client V1 doesn't handle uncaught exceptions in subscription handler functions. If your handler function throws an exception, the subscription closes, and the exception doesn't appear in your component logs. You should catch exceptions in your handler functions to keep the subscription open and log errors that occur in your code.

Publish/subscribe local messages

Publish/subscribe (pubsub) messaging enables you to send and receive messages to topics. Components can publish messages to topics to send messages to other components. Then, components that are subscribed to that topic can act on the messages that they receive.

> **Note**
>
> You can't use this publish/subscribe IPC service to publish or subscribe to AWS IoT Core MQTT. For more information about how to exchange messages with AWS IoT Core MQTT, see Publish/subscribe AWS IoT Core MQTT messages.

Topics

- Minimum SDK versions
- Authorization
- PublishToTopic
- SubscribeToTopic
- Examples

Minimum SDK versions

The following table lists the minimum versions of the SDKs that you can use to publish and subscribe to messages to and from local topics.

SDK	Minimum version	
AWS IoT Greengrass Component SDK (C, C++, Rust)	v1.0.0	
AWS IoT Device SDK for Java v2	v1.2.10	
AWS IoT Device SDK for Python v2	v1.5.3	
AWS IoT Device SDK for C++ v2	v1.17.0	
AWS IoT Device SDK for JavaScript v2	v1.12.0	

Authorization

To use local publish/subscribe messaging in a custom component, you must define authorization policies that allow your component to send and receive messages to topics. For information about defining authorization policies, see Authorize components to perform IPC operations.

Authorization policies for publish/subscribe messaging have the following properties.

IPC service identifier: `aws.greengrass.ipc.pubsub`

Operation	Description	Resources
`aws.greengrass#PublishToTopic`	Allows a component to publish messages to the topics that you specify.	A topic string, such as `test/topic` . Use an `*` to match any combination of characters in a topic. This topic string doesn't support MQTT topic wildcards (# and +).

Operation	Description	Resources
`aws.greengrass#SubscribeToTopic`	Allows a component to subscribe to messages for the topics that you specify.	A topic string, such as `test/topic` . Use an `*` to match any combination of characters in a topic. In Greengrass nucleus v2.6.0 and later, you can subscribe to topics that contain MQTT topic wildcards (# and +). This topic string supports MQTT topic wildcards as literal characters. For example, if a component's authorization policy grants access to `test/topic/#` , the component can subscribe to `test/topic/#` , but it can't subscribe to `test/topic/filter` .

Operation	Description	Resources
*	Allows a component to publish and subscribe to messages for the topics that you specify.	A topic string, such as `test/topic` . Use an * to match any combination of character s in a topic. In Greengrass nucleus v2.6.0 and later, you can subscribe to topics that contain MQTT topic wildcards (# and +). This topic string supports MQTT topic wildcards as literal characters. For example, if a component's authorization policy grants access to `test/topic/#` , the component can subscribe to `test/topic/#` , but it can't subscribe to `test/topic/filter` .

Authorization policy examples

You can reference the following authorization policy example to help you configure authorization policies for your components.

Example authorization policy

The following example authorization policy allows a component to publish and subscribe to all topics.

```
{
  "accessControl": {
    "aws.greengrass.ipc.pubsub": {
      "com.example.MyLocalPubSubComponent:pubsub:1": {
        "policyDescription": "Allows access to publish/subscribe to all topics.",
        "operations": [
          "aws.greengrass#PublishToTopic",
```

```
          "aws.greengrass#SubscribeToTopic"
        ],
        "resources": [
          "*"
        ]
      }
    }
  }
}
```

PublishToTopic

Publish a message to a topic.

Request

This operation's request has the following parameters:

`topic`

The topic to which to publish the message.

`publishMessage` (Python: `publish_message`)

The message to publish. This object, `PublishMessage`, contains the following information. You must specify one of `jsonMessage` and `binaryMessage`.

`jsonMessage` (Python: `json_message`)

(Optional) A JSON message. This object, `JsonMessage`, contains the following information:

`message`

The JSON message as an object.

`context`

The context of the message, such as the topic where the message was published.

This feature is available for v2.6.0 and later of the Greengrass nucleus component. The following table lists the minimum versions of the AWS IoT Device SDK that you must use to access the message context.

SDK	Minimum version	
AWS IoT Device SDK for Java v2	v1.9.3	
AWS IoT Device SDK for Python v2	v1.11.3	
AWS IoT Device SDK for C++ v2	v1.18.4	
AWS IoT Device SDK for JavaScript v2	v1.12.0	

> **Note**
>
> The AWS IoT Greengrass Core software uses the same message objects in the `PublishToTopic` and `SubscribeToTopic` operations. The AWS IoT Greengrass Core software sets this context object in messages when you subscribe, and ignores this context object in messages that you publish.

This object, `MessageContext`, contains the following information:

`topic`

The topic where the message was published.

`binaryMessage` (Python: `binary_message`)

(Optional) A binary message. This object, `BinaryMessage`, contains the following information:

`message`

The binary message as a blob.

`context`

The context of the message, such as the topic where the message was published.

This feature is available for v2.6.0 and later of the Greengrass nucleus component. The following table lists the minimum versions of the AWS IoT Device SDK that you must use to access the message context.

SDK	Minimum version
AWS IoT Device SDK for Java v2	v1.9.3
AWS IoT Device SDK for Python v2	v1.11.3
AWS IoT Device SDK for C++ v2	v1.18.4
AWS IoT Device SDK for JavaScript v2	v1.12.0

Note

The AWS IoT Greengrass Core software uses the same message objects in the `PublishToTopic` and `SubscribeToTopic` operations. The AWS IoT Greengrass Core software sets this context object in messages when you subscribe, and ignores this context object in messages that you publish.

This object, `MessageContext`, contains the following information:

`topic`

The topic where the message was published.

Response

This operation doesn't provide any information in its response.

Examples

The following examples demonstrate how to call this operation in custom component code.

Java (IPC client V2)

Example: Publish a binary message

```
package com.aws.greengrass.docs.samples.ipc;

import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClientV2;
import software.amazon.awssdk.aws.greengrass.model.BinaryMessage;
import software.amazon.awssdk.aws.greengrass.model.PublishMessage;
import software.amazon.awssdk.aws.greengrass.model.PublishToTopicRequest;
import software.amazon.awssdk.aws.greengrass.model.PublishToTopicResponse;
import software.amazon.awssdk.aws.greengrass.model.UnauthorizedError;

import java.nio.charset.StandardCharsets;

public class PublishToTopicV2 {

    public static void main(String[] args) {
        String topic = args[0];
        String message = args[1];
        try (GreengrassCoreIPCClientV2 ipcClient =
 GreengrassCoreIPCClientV2.builder().build()) {
            PublishToTopicV2.publishBinaryMessageToTopic(ipcClient, topic,
 message);
            System.out.println("Successfully published to topic: " + topic);
        } catch (Exception e) {
            if (e.getCause() instanceof UnauthorizedError) {
                System.err.println("Unauthorized error while publishing to topic: "
 + topic);
            } else {
                System.err.println("Exception occurred when using IPC.");
            }
            e.printStackTrace();
            System.exit(1);
        }
    }

    public static PublishToTopicResponse publishBinaryMessageToTopic(
            GreengrassCoreIPCClientV2 ipcClient, String topic, String message)
 throws InterruptedException {
        BinaryMessage binaryMessage =
                new
 BinaryMessage().withMessage(message.getBytes(StandardCharsets.UTF_8));
```

```
        PublishMessage publishMessage = new
 PublishMessage().withBinaryMessage(binaryMessage);
        PublishToTopicRequest publishToTopicRequest =
                new
 PublishToTopicRequest().withTopic(topic).withPublishMessage(publishMessage);
        return ipcClient.publishToTopic(publishToTopicRequest);
    }
}
```

Python (IPC client V2)

Example: Publish a binary message

```
import sys
import traceback

from awsiot.greengrasscoreipc.clientv2 import GreengrassCoreIPCClientV2
from awsiot.greengrasscoreipc.model import (
    PublishMessage,
    BinaryMessage
)


def main():
    args = sys.argv[1:]
    topic = args[0]
    message = args[1]

    try:
        ipc_client = GreengrassCoreIPCClientV2()
        publish_binary_message_to_topic(ipc_client, topic, message)
        print('Successfully published to topic: ' + topic)
    except Exception:
        print('Exception occurred', file=sys.stderr)
        traceback.print_exc()
        exit(1)


def publish_binary_message_to_topic(ipc_client, topic, message):
    binary_message = BinaryMessage(message=bytes(message, 'utf-8'))
    publish_message = PublishMessage(binary_message=binary_message)
    return ipc_client.publish_to_topic(topic=topic,
 publish_message=publish_message)
```

```
if __name__ == '__main__':
    main()
```

C++ (IPC client V1)

Example: Publish a binary message

```
#include <iostream>

#include <aws/crt/Api.h>
#include <aws/greengrass/GreengrassCoreIpcClient.h>

using namespace Aws::Crt;
using namespace Aws::Greengrass;

class IpcClientLifecycleHandler : public ConnectionLifecycleHandler {
    void OnConnectCallback() override {
        // Handle connection to IPC service.
    }

    void OnDisconnectCallback(RpcError error) override {
        // Handle disconnection from IPC service.
    }

    bool OnErrorCallback(RpcError error) override {
        // Handle IPC service connection error.
        return true;
    }
};

int main() {
    ApiHandle apiHandle(g_allocator);
    Io::EventLoopGroup eventLoopGroup(1);
    Io::DefaultHostResolver socketResolver(eventLoopGroup, 64, 30);
    Io::ClientBootstrap bootstrap(eventLoopGroup, socketResolver);
    IpcClientLifecycleHandler ipcLifecycleHandler;
    GreengrassCoreIpcClient ipcClient(bootstrap);
    auto connectionStatus = ipcClient.Connect(ipcLifecycleHandler).get();
    if (!connectionStatus) {
        std::cerr << "Failed to establish IPC connection: " <<
 connectionStatus.StatusToString() << std::endl;
        exit(-1);
    }
```

```
    String topic("my/topic");
    String message("Hello, World!");
    int timeout = 10;

    PublishToTopicRequest request;
    Vector<uint8_t> messageData({message.begin(), message.end()});
    BinaryMessage binaryMessage;
    binaryMessage.SetMessage(messageData);
    PublishMessage publishMessage;
    publishMessage.SetBinaryMessage(binaryMessage);
    request.SetTopic(topic);
    request.SetPublishMessage(publishMessage);

    auto operation = ipcClient.NewPublishToTopic();
    auto activate = operation->Activate(request, nullptr);
    activate.wait();

    auto responseFuture = operation->GetResult();
    if (responseFuture.wait_for(std::chrono::seconds(timeout)) ==
 std::future_status::timeout) {
        std::cerr << "Operation timed out while waiting for response from Greengrass
 Core." << std::endl;
        exit(-1);
    }

    auto response = responseFuture.get();
    if (!response) {
        // Handle error.
        auto errorType = response.GetResultType();
        if (errorType == OPERATION_ERROR) {
            auto *error = response.GetOperationError();
            (void)error;
            // Handle operation error.
        } else {
            // Handle RPC error.
        }
    }
    return 0;
}
```

JavaScript

Example: Publish a binary message

```
import * as greengrasscoreipc from "aws-iot-device-sdk-v2/dist/greengrasscoreipc";
import {BinaryMessage, PublishMessage, PublishToTopicRequest} from "aws-iot-device-
sdk-v2/dist/greengrasscoreipc/model";

class PublishToTopic {
    private ipcClient : greengrasscoreipc.Client
    private readonly topic : string;
    private readonly messageString : string;

    constructor() {
        // define your own constructor, e.g.
        this.topic = "<define_your_topic>";
        this.messageString = "<define_your_message_string>";
        this.publishToTopic().then(r => console.log("Started workflow"));
    }

    private async publishToTopic() {
        try {
            this.ipcClient = await getIpcClient();

            const binaryMessage : BinaryMessage = {
                message: this.messageString
            }

            const publishMessage : PublishMessage = {
                binaryMessage: binaryMessage
            }

            const request : PublishToTopicRequest = {
                topic: this.topic,
                publishMessage: publishMessage
            }

            this.ipcClient.publishToTopic(request).finally(() =>
 console.log(`Published message ${publishMessage.binaryMessage?.message} to topic`))

        } catch (e) {
            // parse the error depending on your use cases
            throw e
```

```
        }
    }
}


export async function getIpcClient(){
    try {
        const ipcClient = greengrasscoreipc.createClient();
        await ipcClient.connect()
            .catch(error => {
                // parse the error depending on your use cases
                throw error;
            });
        return ipcClient
    } catch (err) {
        // parse the error depending on your use cases
        throw err
    }
}

// starting point
const publishToTopic = new PublishToTopic();
```

Rust

Example: Publish a binary message

```
use gg_sdk::Sdk;

fn main() {
    let sdk = Sdk::init();
    sdk.connect().expect("Failed to establish IPC connection");

    let message = b"Hello, World";
    let topic = "my/topic";

    sdk.publish_to_topic_binary(topic, message)
        .expect("Failed to publish to topic");

    println!("Successfully published to topic: {topic}");
}
```

C

Example: Publish a binary message

```
#include <gg/error.h>
#include <gg/ipc/client.h>
#include <gg/sdk.h>
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    gg_sdk_init();

    GgError err = ggipc_connect();
    if (err != GG_ERR_OK) {
        fprintf(stderr, "Failed to establish IPC connection.\n");
        exit(-1);
    }

    GgBuffer message = GG_STR("Hello, World");
    GgBuffer topic = GG_STR("my/topic");

    err = ggipc_publish_to_topic_binary(topic, message);
    if (err != GG_ERR_OK) {
        fprintf(
            stderr,
            "Failed to publish to topic: %.*s\n",
            (int) topic.len,
            topic.data
        );
        exit(-1);
    }

    printf(
        "Successfully published to topic: %.*s\n", (int) topic.len, topic.data
    );
}
```

C++ (Component SDK)

Example: Publish a binary message

```
#include <gg/ipc/client.hpp>
```

```
#include <iostream>

int main() {
    auto &client = gg::ipc::Client::get();

    auto error = client.connect();
    if (error) {
        std::cerr << "Failed to establish IPC connection.\n";
        exit(-1);
    }

    std::string_view message = "Hello, World";
    std::string_view topic = "my/topic";

    error = client.publish_to_topic(topic, message);
    if (error) {
        std::cerr << "Failed to publish to topic: " << topic << "\n";
        exit(-1);
    }

    std::cout << "Successfully published to topic: " << topic << "\n";
}
```

SubscribeToTopic

Subscribe to messages on a topic.

This operation is a subscription operation where you subscribe to a stream of event messages. To use this operation, define a stream response handler with functions that handle event messages, errors, and stream closure. For more information, see Subscribe to IPC event streams.

Event message type: `SubscriptionResponseMessage`

Request

This operation's request has the following parameters:

`topic`

The topic to which to subscribe.

> **Note**
>
> In Greengrass nucleus v2.6.0 and later, this topic supports MQTT topic wildcards (# and +).

`receiveMode` (Python: `receive_mode`)

(Optional) The behavior that specifies whether the component receives messages from itself. You can change this behavior to allow a component to act on its own messages. The default behavior depends on whether the topic contains an MQTT wildcard. Choose from the following options:

- `RECEIVE_ALL_MESSAGES` – Receive all messages that match the topic, including messages from the component that subscribes.

  This mode is the default option when you subscribe to a topic that doesn't contain an MQTT wildcard.

- `RECEIVE_MESSAGES_FROM_OTHERS` – Receive all messages that match the topic, except messages from the component that subscribes.

  This mode is the default option when you subscribe to a topic that contains an MQTT wildcard.

This feature is available for v2.6.0 and later of the Greengrass nucleus component. The following table lists the minimum versions of the AWS IoT Device SDK that you must use to set the receive mode.

SDK	Minimum version
AWS IoT Device SDK for Java v2	v1.9.3
AWS IoT Device SDK for Python v2	v1.11.3
AWS IoT Device SDK for C++ v2	v1.18.4

SDK	Minimum version	
AWS IoT Device SDK for JavaScript v2	v1.12.0	

Response

This operation's response has the following information:

`messages`

The stream of messages. This object, `SubscriptionResponseMessage`, contains the following information. Each message contains `jsonMessage` or `binaryMessage`.

`jsonMessage` (Python: `json_message`)

(Optional) A JSON message. This object, `JsonMessage`, contains the following information:

`message`

The JSON message as an object.

`context`

The context of the message, such as the topic where the message was published.

This feature is available for v2.6.0 and later of the Greengrass nucleus component. The following table lists the minimum versions of the AWS IoT Device SDK that you must use to access the message context.

SDK	Minimum version	
AWS IoT Device SDK for Java v2	v1.9.3	
AWS IoT Device SDK for Python v2	v1.11.3	
AWS IoT Device SDK for C++ v2	v1.18.4	

SDK	Minimum version	
AWS IoT Device SDK for JavaScript v2	v1.12.0	

> **Note**
>
> The AWS IoT Greengrass Core software uses the same message objects in the `PublishToTopic` and `SubscribeToTopic` operations. The AWS IoT Greengrass Core software sets this context object in messages when you subscribe, and ignores this context object in messages that you publish.

This object, `MessageContext`, contains the following information:

`topic`

The topic where the message was published.

`binaryMessage` (Python: `binary_message`)

(Optional) A binary message. This object, `BinaryMessage`, contains the following information:

`message`

The binary message as a blob.

`context`

The context of the message, such as the topic where the message was published.

This feature is available for v2.6.0 and later of the Greengrass nucleus component. The following table lists the minimum versions of the AWS IoT Device SDK that you must use to access the message context.

SDK	Minimum version	
AWS IoT Device SDK for Java v2	v1.9.3	

SDK	Minimum version	
AWS IoT Device SDK for Python v2	v1.11.3	
AWS IoT Device SDK for C++ v2	v1.18.4	
AWS IoT Device SDK for JavaScript v2	v1.12.0	

> **Note**
>
> The AWS IoT Greengrass Core software uses the same message objects in the `PublishToTopic` and `SubscribeToTopic` operations. The AWS IoT Greengrass Core software sets this context object in messages when you subscribe, and ignores this context object in messages that you publish.

This object, `MessageContext`, contains the following information:

`topic`

The topic where the message was published.

`topicName` (Python: `topic_name`)

The topic to which the message was published.

> **Note**
>
> This property isn't currently used. In Greengrass nucleus v2.6.0 and later, you can get the `(jsonMessage|binaryMessage).context.topic` value from a `SubscriptionResponseMessage` to get the topic where the message was published.

Examples

The following examples demonstrate how to call this operation in custom component code.

Java (IPC client V2)

Example: Subscribe to local publish/subscribe messages

```
package com.aws.greengrass.docs.samples.ipc;

import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClientV2;
import software.amazon.awssdk.aws.greengrass.SubscribeToTopicResponseHandler;
import software.amazon.awssdk.aws.greengrass.model.*;

import java.nio.charset.StandardCharsets;
import java.util.Optional;

public class SubscribeToTopicV2 {

    public static void main(String[] args) {
        String topic = args[0];
        try (GreengrassCoreIPCClientV2 ipcClient =
 GreengrassCoreIPCClientV2.builder().build()) {
            SubscribeToTopicRequest request = new
 SubscribeToTopicRequest().withTopic(topic);
            GreengrassCoreIPCClientV2.StreamingResponse<SubscribeToTopicResponse,
                    SubscribeToTopicResponseHandler> response =
                    ipcClient.subscribeToTopic(request,
 SubscribeToTopicV2::onStreamEvent,
                            Optional.of(SubscribeToTopicV2::onStreamError),
                            Optional.of(SubscribeToTopicV2::onStreamClosed));
            SubscribeToTopicResponseHandler responseHandler =
 response.getHandler();
            System.out.println("Successfully subscribed to topic: " + topic);

            // Keep the main thread alive, or the process will exit.
            try {
                while (true) {
                    Thread.sleep(10000);
                }
            } catch (InterruptedException e) {
                System.out.println("Subscribe interrupted.");
            }

            // To stop subscribing, close the stream.
            responseHandler.closeStream();
        } catch (Exception e) {
            if (e.getCause() instanceof UnauthorizedError) {
```

```
                System.err.println("Unauthorized error while publishing to topic: "
 + topic);
            } else {
                System.err.println("Exception occurred when using IPC.");
            }
            e.printStackTrace();
            System.exit(1);
        }
    }

    public static void onStreamEvent(SubscriptionResponseMessage
 subscriptionResponseMessage) {
        try {
            BinaryMessage binaryMessage =
 subscriptionResponseMessage.getBinaryMessage();
            String message = new String(binaryMessage.getMessage(),
 StandardCharsets.UTF_8);
            String topic = binaryMessage.getContext().getTopic();
            System.out.printf("Received new message on topic %s: %s%n", topic,
 message);
        } catch (Exception e) {
            System.err.println("Exception occurred while processing subscription
 response " +
                    "message.");
            e.printStackTrace();
        }
    }

    public static boolean onStreamError(Throwable error) {
        System.err.println("Received a stream error.");
        error.printStackTrace();
        return false; // Return true to close stream, false to keep stream open.
    }

    public static void onStreamClosed() {
        System.out.println("Subscribe to topic stream closed.");
    }
}
```

Python (IPC client V2)

Example: Subscribe to local publish/subscribe messages

```
import sys
```

```
import time
import traceback

from awsiot.greengrasscoreipc.clientv2 import GreengrassCoreIPCClientV2
from awsiot.greengrasscoreipc.model import (
    SubscriptionResponseMessage,
    UnauthorizedError
)


def main():
    args = sys.argv[1:]
    topic = args[0]

    try:
        ipc_client = GreengrassCoreIPCClientV2()
        # Subscription operations return a tuple with the response and the
 operation.
        _, operation = ipc_client.subscribe_to_topic(topic=topic,
 on_stream_event=on_stream_event,

 on_stream_error=on_stream_error, on_stream_closed=on_stream_closed)
        print('Successfully subscribed to topic: ' + topic)

        # Keep the main thread alive, or the process will exit.
        try:
            while True:
                time.sleep(10)
        except InterruptedError:
            print('Subscribe interrupted.')

        # To stop subscribing, close the stream.
        operation.close()
    except UnauthorizedError:
        print('Unauthorized error while subscribing to topic: ' +
              topic, file=sys.stderr)
        traceback.print_exc()
        exit(1)
    except Exception:
        print('Exception occurred', file=sys.stderr)
        traceback.print_exc()
        exit(1)
```

```
def on_stream_event(event: SubscriptionResponseMessage) -> None:
    try:
        message = str(event.binary_message.message, 'utf-8')
        topic = event.binary_message.context.topic
        print('Received new message on topic %s: %s' % (topic, message))
    except:
        traceback.print_exc()


def on_stream_error(error: Exception) -> bool:
    print('Received a stream error.', file=sys.stderr)
    traceback.print_exc()
    return False  # Return True to close stream, False to keep stream open.


def on_stream_closed() -> None:
    print('Subscribe to topic stream closed.')


if __name__ == '__main__':
    main()
```

C++ (IPC client V1)

Example: Subscribe to local publish/subscribe messages

```
#include <iostream>

#include </crt/Api.h>
#include <aws/greengrass/GreengrassCoreIpcClient.h>

using namespace Aws::Crt;
using namespace Aws::Greengrass;

class SubscribeResponseHandler : public SubscribeToTopicStreamHandler {
    public:
        virtual ~SubscribeResponseHandler() {}

    private:
        void OnStreamEvent(SubscriptionResponseMessage *response) override {
            auto jsonMessage = response->GetJsonMessage();
            if (jsonMessage.has_value() &&
 jsonMessage.value().GetMessage().has_value()) {
```

```
                auto messageString =
 jsonMessage.value().GetMessage().value().View().WriteReadable();
                // Handle JSON message.
            } else {
                auto binaryMessage = response->GetBinaryMessage();
                if (binaryMessage.has_value() &&
 binaryMessage.value().GetMessage().has_value()) {
                    auto messageBytes = binaryMessage.value().GetMessage().value();
                    std::string messageString(messageBytes.begin(),
 messageBytes.end());
                    // Handle binary message.
                }
            }
        }

        bool OnStreamError(OperationError *error) override {
            // Handle error.
            return false; // Return true to close stream, false to keep stream open.
        }

        void OnStreamClosed() override {
            // Handle close.
        }
};

class IpcClientLifecycleHandler : public ConnectionLifecycleHandler {
    void OnConnectCallback() override {
        // Handle connection to IPC service.
    }

    void OnDisconnectCallback(RpcError error) override {
        // Handle disconnection from IPC service.
    }

    bool OnErrorCallback(RpcError error) override {
        // Handle IPC service connection error.
        return true;
    }
};

int main() {
    ApiHandle apiHandle(g_allocator);
    Io::EventLoopGroup eventLoopGroup(1);
    Io::DefaultHostResolver socketResolver(eventLoopGroup, 64, 30);
```

```
    Io::ClientBootstrap bootstrap(eventLoopGroup, socketResolver);
    IpcClientLifecycleHandler ipcLifecycleHandler;
    GreengrassCoreIpcClient ipcClient(bootstrap);
    auto connectionStatus = ipcClient.Connect(ipcLifecycleHandler).get();
    if (!connectionStatus) {
        std::cerr << "Failed to establish IPC connection: " <<
connectionStatus.StatusToString() << std::endl;
        exit(-1);
    }

    String topic("my/topic");
    int timeout = 10;

    SubscribeToTopicRequest request;
    request.SetTopic(topic);

    //SubscribeResponseHandler streamHandler;
    auto streamHandler = MakeShared<SubscribeResponseHandler>(DefaultAllocator());
    auto operation = ipcClient.NewSubscribeToTopic(streamHandler);
    auto activate = operation->Activate(request, nullptr);
    activate.wait();

    auto responseFuture = operation->GetResult();
    if (responseFuture.wait_for(std::chrono::seconds(timeout)) ==
std::future_status::timeout) {
        std::cerr << "Operation timed out while waiting for response from Greengrass
Core." << std::endl;
        exit(-1);
    }

    auto response = responseFuture.get();
    if (!response) {
        // Handle error.
        auto errorType = response.GetResultType();
        if (errorType == OPERATION_ERROR) {
            auto *error = response.GetOperationError();
            (void)error;
            // Handle operation error.
        } else {
            // Handle RPC error.
        }
        exit(-1);
    }
```

```
    // Keep the main thread alive, or the process will exit.
    while (true) {
        std::this_thread::sleep_for(std::chrono::seconds(10));
    }

    operation->Close();
    return 0;
}
```

JavaScript

Example: Subscribe to local publish/subscribe messages

```
import * as greengrasscoreipc from "aws-iot-device-sdk-v2/dist/greengrasscoreipc";
import {SubscribeToTopicRequest, SubscriptionResponseMessage} from "aws-iot-device-
sdk-v2/dist/greengrasscoreipc/model";
import {RpcError} from "aws-iot-device-sdk-v2/dist/eventstream_rpc";

class SubscribeToTopic {
    private ipcClient : greengrasscoreipc.Client
    private readonly topic : string;

    constructor() {
        // define your own constructor, e.g.
        this.topic = "<define_your_topic>";
        this.subscribeToTopic().then(r => console.log("Started workflow"));
    }

    private async subscribeToTopic() {
        try {
            this.ipcClient = await getIpcClient();

            const subscribeToTopicRequest : SubscribeToTopicRequest = {
                topic: this.topic,
            }

            const streamingOperation =
 this.ipcClient.subscribeToTopic(subscribeToTopicRequest, undefined); //
 conditionally apply options

            streamingOperation.on("message", (message: SubscriptionResponseMessage)
 => {
                // parse the message depending on your use cases, e.g.
```

```
                if(message.binaryMessage && message.binaryMessage.message) {
                    const receivedMessage =
 message.binaryMessage?.message.toString();
                }
            });

            streamingOperation.on("streamError", (error : RpcError) => {
                // define your own error handling logic
            })

            streamingOperation.on("ended", () => {
                // define your own logic
            })

            await streamingOperation.activate();

            // Keep the main thread alive, or the process will exit.
            await new Promise((resolve) => setTimeout(resolve, 10000))
        } catch (e) {
            // parse the error depending on your use cases
            throw e
        }
    }
}

export async function getIpcClient(){
    try {
        const ipcClient = greengrasscoreipc.createClient();
        await ipcClient.connect()
            .catch(error => {
                // parse the error depending on your use cases
                throw error;
            });
        return ipcClient
    } catch (err) {
        // parse the error depending on your use cases
        throw err
    }
}

// starting point
const subscribeToTopic = new SubscribeToTopic();
```

Rust

Example: Subscribe to local publish/subscribe messages

```
use gg_sdk::{Sdk, SubscribeToTopicPayload};
use std::{thread, time::Duration};

fn main() {
    let sdk = Sdk::init();
    sdk.connect().expect("Failed to establish IPC connection");

    let topic = "my/topic";

    let callback = |topic: &str, payload: SubscribeToTopicPayload| match payload
    {
        SubscribeToTopicPayload::Binary(message) => {
            let message = String::from_utf8_lossy(message);
            println!("Received new message on topic {topic}: {message}");
        }
        SubscribeToTopicPayload::Json(_) => {
            println!("Received new message on topic {topic}: (JSON message)");
        }
    };

    let _sub = sdk
        .subscribe_to_topic(topic, &callback)
        .expect("Failed to subscribe to topic");

    println!("Successfully subscribed to topic: {topic}");

    // Keep the main thread alive, or the process will exit.
    loop {
        thread::sleep(Duration::from_secs(10));
    }
}
```

C

Example: Subscribe to local publish/subscribe messages

```
#include <assert.h>
#include <gg/error.h>
#include <gg/ipc/client.h>
#include <gg/object.h>
```

```
#include <gg/sdk.h>
#include <gg/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

static void on_subscription_response(
    void *ctx, GgBuffer topic, GgObject payload, GgIpcSubscriptionHandle handle
) {
    (void) ctx;
    (void) handle;

    if (gg_obj_type(payload) == GG_TYPE_BUF) {
        GgBuffer message = gg_obj_into_buf(payload);
        printf(
            "Received new message on topic %.*s: %.*s\n",
            (int) topic.len,
            topic.data,
            (int) message.len,
            message.data
        );
    } else {
        assert(gg_obj_type(payload) == GG_TYPE_MAP);
        printf(
            "Received new message on topic %.*s: (JSON message)\n",
            (int) topic.len,
            topic.data
        );
    }
}

int main(void) {
    gg_sdk_init();

    GgError err = ggipc_connect();
    if (err != GG_ERR_OK) {
        fprintf(stderr, "Failed to establish IPC connection.\n");
        exit(-1);
    }

    GgBuffer topic = GG_STR("my/topic");

    GgIpcSubscriptionHandle handle;
    err = ggipc_subscribe_to_topic(
```

```
        topic, on_subscription_response, NULL, &handle
    );
    if (err != GG_ERR_OK) {
        fprintf(
            stderr,
            "Failed to subscribe to topic: %.*s\n",
            (int) topic.len,
            topic.data
        );
        exit(-1);
    }

    printf(
        "Successfully subscribed to topic: %.*s\n", (int) topic.len, topic.data
    );

    // Keep the main thread alive, or the process will exit.
    while (1) {
        sleep(10);
    }

    // To stop subscribing, close the stream.
    ggipc_close_subscription(handle);
}
```

C++ (Component SDK)

Example: Subscribe to local publish/subscribe messages

```
#include <gg/ipc/client.hpp>
#include <gg/object.hpp>
#include <unistd.h>
#include <cassert>
#include <iostream>

class ResponseHandler : public gg::ipc::LocalTopicCallback {
    void operator()(
        std::string_view topic,
        gg::Object payload,
        gg::ipc::Subscription &handle
    ) override {
        (void) handle;
        if (payload.index() == GG_TYPE_BUF) {
```

```
            std::cout << "Received new message on topic " << topic << ": "
                      << get<gg::Buffer>(payload) << "\n";
        } else {
            assert(payload.index() == GG_TYPE_MAP);
            std::cout << "Received new message on topic " << topic
                      << ": (JSON message)\n";
        }
    }
};

int main() {
    auto &client = gg::ipc::Client::get();

    auto error = client.connect();
    if (error) {
        std::cerr << "Failed to establish IPC connection.\n";
        exit(-1);
    }

    std::string_view topic = "my/topic";

    static ResponseHandler handler;
    error = client.subscribe_to_topic(topic, handler);
    if (error) {
        std::cerr << "Failed to subscribe to topic: " << topic << "\n";
        exit(-1);
    }

    std::cout << "Successfully subscribed to topic: " << topic << "\n";

    // Keep the main thread alive, or the process will exit.
    while (1) {
        sleep(10);
    }
}
```

Examples

Use the following examples to learn how to use the publish/subscribe IPC service in your components.

Example publish/subscribe publisher (Java, IPC client V1)

The following example recipe allows the component to publish to all topics.

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.PubSubPublisherJava",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that publishes messages.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
          "com.example.PubSubPublisherJava:pubsub:1": {
            "policyDescription": "Allows access to publish to all topics.",
            "operations": [
              "aws.greengrass#PublishToTopic"
            ],
            "resources": [
              "*"
            ]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Lifecycle": {
        "Run": "java -jar {artifacts:path}/PubSubPublisher.jar"
      }
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.PubSubPublisherJava
```

```
ComponentVersion: '1.0.0'
ComponentDescription: A component that publishes messages.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    accessControl:
      aws.greengrass.ipc.pubsub:
        'com.example.PubSubPublisherJava:pubsub:1':
          policyDescription: Allows access to publish to all topics.
          operations:
            - 'aws.greengrass#PublishToTopic'
          resources:
            - '*'
Manifests:
  - Lifecycle:
      Run: |-
        java -jar {artifacts:path}/PubSubPublisher.jar
```

The following example Java application demonstrates how to use the publish/subscribe IPC service to publish messages to other components.

```
/* Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
 * SPDX-License-Identifier: Apache-2.0 */

package com.example.ipc.pubsub;

import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClient;
import software.amazon.awssdk.aws.greengrass.model.*;
import software.amazon.awssdk.eventstreamrpc.EventStreamRPCConnection;

import java.nio.charset.StandardCharsets;
import java.util.Optional;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public class PubSubPublisher {

    public static void main(String[] args) {
        String message = "Hello from the pub/sub publisher (Java).";
        String topic = "test/topic/java";
```

```
        try (EventStreamRPCConnection eventStreamRPCConnection =
IPCUtils.getEventStreamRpcConnection()) {
            GreengrassCoreIPCClient ipcClient = new
GreengrassCoreIPCClient(eventStreamRPCConnection);

            while (true) {
                PublishToTopicRequest publishRequest = new PublishToTopicRequest();
                PublishMessage publishMessage = new PublishMessage();
                BinaryMessage binaryMessage = new BinaryMessage();
                binaryMessage.setMessage(message.getBytes(StandardCharsets.UTF_8));
                publishMessage.setBinaryMessage(binaryMessage);
                publishRequest.setPublishMessage(publishMessage);
                publishRequest.setTopic(topic);
                CompletableFuture<PublishToTopicResponse> futureResponse = ipcClient
                        .publishToTopic(publishRequest,
Optional.empty()).getResponse();

                try {
                    futureResponse.get(10, TimeUnit.SECONDS);
                    System.out.println("Successfully published to topic: " + topic);
                } catch (TimeoutException e) {
                    System.err.println("Timeout occurred while publishing to topic: " +
topic);
                } catch (ExecutionException e) {
                    if (e.getCause() instanceof UnauthorizedError) {
                        System.err.println("Unauthorized error while publishing to
topic: " + topic);
                    } else {
                        System.err.println("Execution exception while publishing to
topic: " + topic);
                    }
                    throw e;
                }
                Thread.sleep(5000);
            }
        } catch (InterruptedException e) {
            System.out.println("Publisher interrupted.");
        } catch (Exception e) {
            System.err.println("Exception occurred when using IPC.");
            e.printStackTrace();
            System.exit(1);
        }
    }
```

```
}
```

Example publish/subscribe subscriber (Java, IPC client V1)

The following example recipe allows the component to subscribe to all topics.

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.PubSubSubscriberJava",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that subscribes to messages.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
          "com.example.PubSubSubscriberJava:pubsub:1": {
            "policyDescription": "Allows access to subscribe to all topics.",
            "operations": [
              "aws.greengrass#SubscribeToTopic"
            ],
            "resources": [
              "*"
            ]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Lifecycle": {
        "Run": "java -jar {artifacts:path}/PubSubSubscriber.jar"
      }
    }
  ]
}
```

YAML

```
---
```

```
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.PubSubSubscriberJava
ComponentVersion: '1.0.0'
ComponentDescription: A component that subscribes to messages.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    accessControl:
      aws.greengrass.ipc.pubsub:
        'com.example.PubSubSubscriberJava:pubsub:1':
          policyDescription: Allows access to subscribe to all topics.
          operations:
            - 'aws.greengrass#SubscribeToTopic'
          resources:
            - '*'
Manifests:
  - Lifecycle:
      Run: |-
        java -jar {artifacts:path}/PubSubSubscriber.jar
```

The following example Java application demonstrates how to use the publish/subscribe IPC service to subscribe to messages to other components.

```
/* Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
 * SPDX-License-Identifier: Apache-2.0 */

package com.example.ipc.pubsub;

import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClient;
import software.amazon.awssdk.aws.greengrass.SubscribeToTopicResponseHandler;
import software.amazon.awssdk.aws.greengrass.model.SubscribeToTopicRequest;
import software.amazon.awssdk.aws.greengrass.model.SubscribeToTopicResponse;
import software.amazon.awssdk.aws.greengrass.model.SubscriptionResponseMessage;
import software.amazon.awssdk.aws.greengrass.model.UnauthorizedError;
import software.amazon.awssdk.eventstreamrpc.EventStreamRPCConnection;
import software.amazon.awssdk.eventstreamrpc.StreamResponseHandler;

import java.nio.charset.StandardCharsets;
import java.util.Optional;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;
```

```
import java.util.concurrent.TimeoutException;

public class PubSubSubscriber {

    public static void main(String[] args) {
        String topic = "test/topic/java";

        try (EventStreamRPCConnection eventStreamRPCConnection =
 IPCUtils.getEventStreamRpcConnection()) {
            GreengrassCoreIPCClient ipcClient = new
 GreengrassCoreIPCClient(eventStreamRPCConnection);

            SubscribeToTopicRequest subscribeRequest = new SubscribeToTopicRequest();
            subscribeRequest.setTopic(topic);
            SubscribeToTopicResponseHandler operationResponseHandler = ipcClient
                    .subscribeToTopic(subscribeRequest, Optional.of(new
 SubscribeResponseHandler()));
            CompletableFuture<SubscribeToTopicResponse> futureResponse =
 operationResponseHandler.getResponse();

            try {
                futureResponse.get(10, TimeUnit.SECONDS);
                System.out.println("Successfully subscribed to topic: " + topic);
            } catch (TimeoutException e) {
                System.err.println("Timeout occurred while subscribing to topic: " +
 topic);
                throw e;
            } catch (ExecutionException e) {
                if (e.getCause() instanceof UnauthorizedError) {
                    System.err.println("Unauthorized error while subscribing to topic:
 " + topic);
                } else {
                    System.err.println("Execution exception while subscribing to topic:
 " + topic);
                }
                throw e;
            }

            // Keep the main thread alive, or the process will exit.
            try {
                while (true) {
                    Thread.sleep(10000);
                }
            } catch (InterruptedException e) {
```

```
                System.out.println("Subscribe interrupted.");
            }
        } catch (Exception e) {
            System.err.println("Exception occurred when using IPC.");
            e.printStackTrace();
            System.exit(1);
        }
    }

    private static class SubscribeResponseHandler implements
 StreamResponseHandler<SubscriptionResponseMessage> {

        @Override
        public void onStreamEvent(SubscriptionResponseMessage
 subscriptionResponseMessage) {
            try {
                String message = new
 String(subscriptionResponseMessage.getBinaryMessage()
                        .getMessage(), StandardCharsets.UTF_8);
                System.out.println("Received new message: " + message);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        @Override
        public boolean onStreamError(Throwable error) {
            System.err.println("Received a stream error.");
            error.printStackTrace();
            return false; // Return true to close stream, false to keep stream open.
        }

        @Override
        public void onStreamClosed() {
            System.out.println("Subscribe to topic stream closed.");
        }
    }
}
```

Example publish/subscribe publisher (Python, IPC client V1)

The following example recipe allows the component to publish to all topics.

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.PubSubPublisherPython",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that publishes messages.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
          "com.example.PubSubPublisherPython:pubsub:1": {
            "policyDescription": "Allows access to publish to all topics.",
            "operations": [
              "aws.greengrass#PublishToTopic"
            ],
            "resources": [
              "*"
            ]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "install": "python3 -m pip install --user awsiotsdk",
        "Run": "python3 -u {artifacts:path}/pubsub_publisher.py"
      }
    },
    {
      "Platform": {
        "os": "windows"
      },
      "Lifecycle": {
        "install": "py -3 -m pip install --user awsiotsdk",
        "Run": "py -3 -u {artifacts:path}/pubsub_publisher.py"
      }
    }
```

```
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.PubSubPublisherPython
ComponentVersion: 1.0.0
ComponentDescription: A component that publishes messages.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    accessControl:
      aws.greengrass.ipc.pubsub:
        com.example.PubSubPublisherPython:pubsub:1:
          policyDescription: Allows access to publish to all topics.
          operations:
            - aws.greengrass#PublishToTopic
          resources:
            - "*"
Manifests:
  - Platform:
      os: linux
    Lifecycle:
      install: python3 -m pip install --user awsiotsdk
      Run: python3 -u {artifacts:path}/pubsub_publisher.py
  - Platform:
      os: windows
    Lifecycle:
      install: py -3 -m pip install --user awsiotsdk
      Run: py -3 -u {artifacts:path}/pubsub_publisher.py
```

The following example Python application demonstrates how to use the publish/subscribe IPC service to publish messages to other components.

```
import concurrent.futures
import sys
import time
import traceback

import awsiot.greengrasscoreipc
```

```
from awsiot.greengrasscoreipc.model import (
    PublishToTopicRequest,
    PublishMessage,
    BinaryMessage,
    UnauthorizedError
)


topic = "test/topic/python"
message = "Hello from the pub/sub publisher (Python)."
TIMEOUT = 10

try:
    ipc_client = awsiot.greengrasscoreipc.connect()

    while True:
        request = PublishToTopicRequest()
        request.topic = topic
        publish_message = PublishMessage()
        publish_message.binary_message = BinaryMessage()
        publish_message.binary_message.message = bytes(message, "utf-8")
        request.publish_message = publish_message
        operation = ipc_client.new_publish_to_topic()
        operation.activate(request)
        future_response = operation.get_response()

        try:
            future_response.result(TIMEOUT)
            print('Successfully published to topic: ' + topic)
        except concurrent.futures.TimeoutError:
            print('Timeout occurred while publishing to topic: ' + topic,
 file=sys.stderr)
        except UnauthorizedError as e:
            print('Unauthorized error while publishing to topic: ' + topic,
 file=sys.stderr)
            raise e
        except Exception as e:
            print('Exception while publishing to topic: ' + topic, file=sys.stderr)
            raise e
        time.sleep(5)
except InterruptedError:
    print('Publisher interrupted.')
except Exception:
    print('Exception occurred when using IPC.', file=sys.stderr)
```

```
    traceback.print_exc()
    exit(1)
```

Example publish/subscribe subscriber (Python, IPC client V1)

The following example recipe allows the component to subscribe to all topics.

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.PubSubSubscriberPython",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that subscribes to messages.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
          "com.example.PubSubSubscriberPython:pubsub:1": {
            "policyDescription": "Allows access to subscribe to all topics.",
            "operations": [
              "aws.greengrass#SubscribeToTopic"
            ],
            "resources": [
              "*"
            ]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "install": "python3 -m pip install --user awsiotsdk",
        "Run": "python3 -u {artifacts:path}/pubsub_subscriber.py"
      }
    },
    {
      "Platform": {
```

```
        "os": "windows"
      },
      "Lifecycle": {
        "install": "py -3 -m pip install --user awsiotsdk",
        "Run": "py -3 -u {artifacts:path}/pubsub_subscriber.py"
      }
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.PubSubSubscriberPython
ComponentVersion: 1.0.0
ComponentDescription: A component that subscribes to messages.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    accessControl:
      aws.greengrass.ipc.pubsub:
        com.example.PubSubSubscriberPython:pubsub:1:
          policyDescription: Allows access to subscribe to all topics.
          operations:
            - aws.greengrass#SubscribeToTopic
          resources:
            - "*"
Manifests:
  - Platform:
      os: linux
    Lifecycle:
      install: python3 -m pip install --user awsiotsdk
      Run: python3 -u {artifacts:path}/pubsub_subscriber.py
  - Platform:
      os: windows
    Lifecycle:
      install: py -3 -m pip install --user awsiotsdk
      Run: py -3 -u {artifacts:path}/pubsub_subscriber.py
```

The following example Python application demonstrates how to use the publish/subscribe IPC service to subscribe to messages to other components.

```
import concurrent.futures
import sys
import time
import traceback

import awsiot.greengrasscoreipc
import awsiot.greengrasscoreipc.client as client
from awsiot.greengrasscoreipc.model import (
    SubscribeToTopicRequest,
    SubscriptionResponseMessage,
    UnauthorizedError
)

topic = "test/topic/python"
TIMEOUT = 10


class StreamHandler(client.SubscribeToTopicStreamHandler):
    def __init__(self):
        super().__init__()

    def on_stream_event(self, event: SubscriptionResponseMessage) -> None:
        try:
            message = str(event.binary_message.message, "utf-8")
            print("Received new message: " + message)
        except:
            traceback.print_exc()

    def on_stream_error(self, error: Exception) -> bool:
        print("Received a stream error.", file=sys.stderr)
        traceback.print_exc()
        return False  # Return True to close stream, False to keep stream open.

    def on_stream_closed(self) -> None:
        print('Subscribe to topic stream closed.')


try:
    ipc_client = awsiot.greengrasscoreipc.connect()

    request = SubscribeToTopicRequest()
    request.topic = topic
    handler = StreamHandler()
```

```
    operation = ipc_client.new_subscribe_to_topic(handler)
    operation.activate(request)
    future_response = operation.get_response()

    try:
        future_response.result(TIMEOUT)
        print('Successfully subscribed to topic: ' + topic)
    except concurrent.futures.TimeoutError as e:
        print('Timeout occurred while subscribing to topic: ' + topic,
 file=sys.stderr)
        raise e
    except UnauthorizedError as e:
        print('Unauthorized error while subscribing to topic: ' + topic,
 file=sys.stderr)
        raise e
    except Exception as e:
        print('Exception while subscribing to topic: ' + topic, file=sys.stderr)
        raise e

    # Keep the main thread alive, or the process will exit.
    try:
        while True:
            time.sleep(10)
    except InterruptedError:
        print('Subscribe interrupted.')
except Exception:
    print('Exception occurred when using IPC.', file=sys.stderr)
    traceback.print_exc()
    exit(1)
```

Example publish/subscribe publisher (C++, IPC client V1)

The following example recipe allows the component to publish to all topics.

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.PubSubPublisherCpp",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that publishes messages.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
```

```
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
          "com.example.PubSubPublisherCpp:pubsub:1": {
            "policyDescription": "Allows access to publish to all topics.",
            "operations": [
              "aws.greengrass#PublishToTopic"
            ],
            "resources": [
              "*"
            ]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Lifecycle": {
        "Run": "{artifacts:path}/greengrassv2_pubsub_publisher"
      },
      "Artifacts": [
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.PubSubPublisherCpp/1.0.0/greengrassv2_pubsub_publisher",
          "Permission": {
            "Execute": "OWNER"
          }
        }
      ]
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.PubSubPublisherCpp
ComponentVersion: 1.0.0
ComponentDescription: A component that publishes messages.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
```

```
    accessControl:
      aws.greengrass.ipc.pubsub:
        com.example.PubSubPublisherCpp:pubsub:1:
          policyDescription: Allows access to publish to all topics.
          operations:
            - aws.greengrass#PublishToTopic
          resources:
            - "*"
Manifests:
  - Lifecycle:
      Run: "{artifacts:path}/greengrassv2_pubsub_publisher"
    Artifacts:
      - URI: s3://amzn-s3-demo-bucket/artifacts/
com.example.PubSubPublisherCpp/1.0.0/greengrassv2_pubsub_publisher
        Permission:
          Execute: OWNER
```

The following example C++ application demonstrates how to use the publish/subscribe IPC service to publish messages to other components.

```
#include <iostream>

#include <aws/crt/Api.h>
#include <aws/greengrass/GreengrassCoreIpcClient.h>

using namespace Aws::Crt;
using namespace Aws::Greengrass;

class IpcClientLifecycleHandler : public ConnectionLifecycleHandler {
    void OnConnectCallback() override {
        std::cout << "OnConnectCallback" << std::endl;
    }

    void OnDisconnectCallback(RpcError error) override {
        std::cout << "OnDisconnectCallback: " << error.StatusToString() << std::endl;
        exit(-1);
    }

    bool OnErrorCallback(RpcError error) override {
        std::cout << "OnErrorCallback: " << error.StatusToString() << std::endl;
        return true;
    }
```

```
};

int main() {
    String message("Hello from the pub/sub publisher (C++).");
    String topic("test/topic/cpp");
    int timeout = 10;

    ApiHandle apiHandle(g_allocator);
    Io::EventLoopGroup eventLoopGroup(1);
    Io::DefaultHostResolver socketResolver(eventLoopGroup, 64, 30);
    Io::ClientBootstrap bootstrap(eventLoopGroup, socketResolver);
    IpcClientLifecycleHandler ipcLifecycleHandler;
    GreengrassCoreIpcClient ipcClient(bootstrap);
    auto connectionStatus = ipcClient.Connect(ipcLifecycleHandler).get();
    if (!connectionStatus) {
        std::cerr << "Failed to establish IPC connection: " <<
 connectionStatus.StatusToString() << std::endl;
        exit(-1);
    }

    while (true) {
        PublishToTopicRequest request;
        Vector<uint8_t> messageData({message.begin(), message.end()});
        BinaryMessage binaryMessage;
        binaryMessage.SetMessage(messageData);
        PublishMessage publishMessage;
        publishMessage.SetBinaryMessage(binaryMessage);
        request.SetTopic(topic);
        request.SetPublishMessage(publishMessage);

        auto operation = ipcClient.NewPublishToTopic();
        auto activate = operation->Activate(request, nullptr);
        activate.wait();

        auto responseFuture = operation->GetResult();
        if (responseFuture.wait_for(std::chrono::seconds(timeout)) ==
 std::future_status::timeout) {
            std::cerr << "Operation timed out while waiting for response from
 Greengrass Core." << std::endl;
            exit(-1);
        }

        auto response = responseFuture.get();
        if (response) {
```

```
            std::cout << "Successfully published to topic: " << topic << std::endl;
        } else {
            // An error occurred.
            std::cout << "Failed to publish to topic: " << topic << std::endl;
            auto errorType = response.GetResultType();
            if (errorType == OPERATION_ERROR) {
                auto *error = response.GetOperationError();
                std::cout << "Operation error: " << error->GetMessage().value() <<
 std::endl;
            } else {
                std::cout << "RPC error: " << response.GetRpcError() << std::endl;
            }
            exit(-1);
        }

        std::this_thread::sleep_for(std::chrono::seconds(5));
    }

    return 0;
}
```

Example publish/subscribe subscriber (C++, IPC client V1)

The following example recipe allows the component to subscribe to all topics.

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.PubSubSubscriberCpp",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that subscribes to messages.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
          "com.example.PubSubSubscriberCpp:pubsub:1": {
            "policyDescription": "Allows access to subscribe to all topics.",
            "operations": [
              "aws.greengrass#SubscribeToTopic"
            ],
            "resources": [
              "*"
```

```
            ]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Lifecycle": {
        "Run": "{artifacts:path}/greengrassv2_pub_sub_subscriber"
      },
      "Artifacts": [
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.PubSubSubscriberCpp/1.0.0/greengrassv2_pub_sub_subscriber",
          "Permission": {
            "Execute": "OWNER"
          }
        }
      ]
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.PubSubSubscriberCpp
ComponentVersion: 1.0.0
ComponentDescription: A component that subscribes to messages.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    accessControl:
      aws.greengrass.ipc.pubsub:
        com.example.PubSubSubscriberCpp:pubsub:1:
          policyDescription: Allows access to subscribe to all topics.
          operations:
            - aws.greengrass#SubscribeToTopic
          resources:
            - "*"
Manifests:
```

```
  - Lifecycle:
      Run: "{artifacts:path}/greengrassv2_pub_sub_subscriber"
    Artifacts:
      - URI: s3://amzn-s3-demo-bucket/artifacts/
com.example.PubSubSubscriberCpp/1.0.0/greengrassv2_pub_sub_subscriber
        Permission:
          Execute: OWNER
```

The following example C++ application demonstrates how to use the publish/subscribe IPC service to subscribe to messages to other components.

```
#include <iostream>

#include <aws/crt/Api.h>
#include <aws/greengrass/GreengrassCoreIpcClient.h>

using namespace Aws::Crt;
using namespace Aws::Greengrass;

class SubscribeResponseHandler : public SubscribeToTopicStreamHandler {
    public:
        virtual ~SubscribeResponseHandler() {}

    private:
        void OnStreamEvent(SubscriptionResponseMessage *response) override {
            auto jsonMessage = response->GetJsonMessage();
            if (jsonMessage.has_value() &&
 jsonMessage.value().GetMessage().has_value()) {
                auto messageString =
 jsonMessage.value().GetMessage().value().View().WriteReadable();
                std::cout << "Received new message: " << messageString << std::endl;
            } else {
                auto binaryMessage = response->GetBinaryMessage();
                if (binaryMessage.has_value() &&
 binaryMessage.value().GetMessage().has_value()) {
                    auto messageBytes = binaryMessage.value().GetMessage().value();
                    std::string messageString(messageBytes.begin(),
 messageBytes.end());
                    std::cout << "Received new message: " << messageString <<
 std::endl;
                }
            }
```

```
        }

        bool OnStreamError(OperationError *error) override {
            std::cout << "Received an operation error: ";
            if (error->GetMessage().has_value()) {
                std::cout << error->GetMessage().value();
            }
            std::cout << std::endl;
            return false; // Return true to close stream, false to keep stream open.
        }

        void OnStreamClosed() override {
            std::cout << "Subscribe to topic stream closed." << std::endl;
        }
};

class IpcClientLifecycleHandler : public ConnectionLifecycleHandler {
    void OnConnectCallback() override {
        std::cout << "OnConnectCallback" << std::endl;
    }

    void OnDisconnectCallback(RpcError error) override {
        std::cout << "OnDisconnectCallback: " << error.StatusToString() << std::endl;
        exit(-1);
    }

    bool OnErrorCallback(RpcError error) override {
        std::cout << "OnErrorCallback: " << error.StatusToString() << std::endl;
        return true;
    }
};

int main() {
    String topic("test/topic/cpp");
    int timeout = 10;

    ApiHandle apiHandle(g_allocator);
    Io::EventLoopGroup eventLoopGroup(1);
    Io::DefaultHostResolver socketResolver(eventLoopGroup, 64, 30);
    Io::ClientBootstrap bootstrap(eventLoopGroup, socketResolver);
    IpcClientLifecycleHandler ipcLifecycleHandler;
    GreengrassCoreIpcClient ipcClient(bootstrap);
    auto connectionStatus = ipcClient.Connect(ipcLifecycleHandler).get();
    if (!connectionStatus) {
```

```
        std::cerr << "Failed to establish IPC connection: " <<
connectionStatus.StatusToString() << std::endl;
        exit(-1);
    }

    SubscribeToTopicRequest request;
    request.SetTopic(topic);
    auto streamHandler = MakeShared<SubscribeResponseHandler>(DefaultAllocator());
    auto operation = ipcClient.NewSubscribeToTopic(streamHandler);
    auto activate = operation->Activate(request, nullptr);
    activate.wait();

    auto responseFuture = operation->GetResult();
    if (responseFuture.wait_for(std::chrono::seconds(timeout)) ==
std::future_status::timeout) {
        std::cerr << "Operation timed out while waiting for response from Greengrass
Core." << std::endl;
        exit(-1);
    }

    auto response = responseFuture.get();
    if (response) {
        std::cout << "Successfully subscribed to topic: " << topic << std::endl;
    } else {
        // An error occurred.
        std::cout << "Failed to subscribe to topic: " << topic << std::endl;
        auto errorType = response.GetResultType();
        if (errorType == OPERATION_ERROR) {
            auto *error = response.GetOperationError();
            std::cout << "Operation error: " << error->GetMessage().value() <<
std::endl;
        } else {
            std::cout << "RPC error: " << response.GetRpcError() << std::endl;
        }
        exit(-1);
    }

    // Keep the main thread alive, or the process will exit.
    while (true) {
        std::this_thread::sleep_for(std::chrono::seconds(10));
    }

    operation->Close();
    return 0;
```

```
}
```

Example publish/subscribe publisher (Rust)

The following example recipe allows the component to publish to all topics.

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.PubSubPublisherRust",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that publishes messages.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
          "com.example.PubSubPublisherRust:pubsub:1": {
            "policyDescription": "Allows access to publish to all topics.",
            "operations": ["aws.greengrass#PublishToTopic"],
            "resources": ["*"]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux",
        "runtime": "*"
      },
      "Lifecycle": {
        "run": "{artifacts:path}/publish_to_topic"
      },
      "Artifacts": [
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.PubSubPublisherRust/1.0.0/publish_to_topic",
          "Permission": {
            "Execute": "OWNER"
          }
        }
      ]
    }
```

```
  ]
}
```

The following example Rust application demonstrates how to use the publish/subscribe IPC service to publish messages to other components.

```
use gg_sdk::Sdk;

fn main() {
    let sdk = Sdk::init();
    sdk.connect().expect("Failed to establish IPC connection");

    let message = b"Hello, World";
    let topic = "my/topic";

    sdk.publish_to_topic_binary(topic, message)
        .expect("Failed to publish to topic");

    println!("Successfully published to topic: {topic}");
}
```

Example publish/subscribe subscriber (Rust)

The following example recipe allows the component to subscribe to all topics.

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.PubSubSubscriberRust",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that subscribes to messages.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
          "com.example.PubSubSubscriberRust:pubsub:1": {
            "policyDescription": "Allows access to subscribe to all topics.",
            "operations": ["aws.greengrass#SubscribeToTopic"],
            "resources": ["*"]
          }
        }
      }
    }
```

```
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux",
        "runtime": "*"
      },
      "Lifecycle": {
        "run": "{artifacts:path}/subscribe_to_topic"
      },
      "Artifacts": [
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.PubSubSubscriberRust/1.0.0/subscribe_to_topic",
          "Permission": {
            "Execute": "OWNER"
          }
        }
      ]
    }
  ]
}
```

The following example Rust application demonstrates how to use the publish/subscribe IPC service to subscribe to messages from other components.

```
use gg_sdk::{Sdk, SubscribeToTopicPayload};
use std::{thread, time::Duration};

fn main() {
    let sdk = Sdk::init();
    sdk.connect().expect("Failed to establish IPC connection");

    let topic = "my/topic";

    let callback = |topic: &str, payload: SubscribeToTopicPayload| match payload
    {
        SubscribeToTopicPayload::Binary(message) => {
            let message = String::from_utf8_lossy(message);
            println!("Received new message on topic {topic}: {message}");
        }
        SubscribeToTopicPayload::Json(_) => {
            println!("Received new message on topic {topic}: (JSON message)");
```

```
        }
    };

    let _sub = sdk
        .subscribe_to_topic(topic, &callback)
        .expect("Failed to subscribe to topic");

    println!("Successfully subscribed to topic: {topic}");

    // Keep the main thread alive, or the process will exit.
    loop {
        thread::sleep(Duration::from_secs(10));
    }
}
```

Example publish/subscribe publisher (C)

The following example recipe allows the component to publish to all topics.

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.PubSubPublisherC",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that publishes messages.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
          "com.example.PubSubPublisherC:pubsub:1": {
            "policyDescription": "Allows access to publish to all topics.",
            "operations": ["aws.greengrass#PublishToTopic"],
            "resources": ["*"]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux",
        "runtime": "*"
      },
```

```
      "Lifecycle": {
        "run": "{artifacts:path}/sample_publish_to_topic"
      },
      "Artifacts": [
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.PubSubPublisherC/1.0.0/sample_publish_to_topic",
          "Permission": {
            "Execute": "OWNER"
          }
        }
      ]
    }
  ]
}
```

The following example C application demonstrates how to use the publish/subscribe IPC service to publish messages to other components.

```
#include <gg/error.h>
#include <gg/ipc/client.h>
#include <gg/sdk.h>
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    gg_sdk_init();

    GgError err = ggipc_connect();
    if (err != GG_ERR_OK) {
        fprintf(stderr, "Failed to establish IPC connection.\n");
        exit(-1);
    }

    GgBuffer message = GG_STR("Hello, World");
    GgBuffer topic = GG_STR("my/topic");

    err = ggipc_publish_to_topic_binary(topic, message);
    if (err != GG_ERR_OK) {
        fprintf(
            stderr,
            "Failed to publish to topic: %.*s\n",
            (int) topic.len,
```

```
            topic.data
        );
        exit(-1);
    }

    printf(
        "Successfully published to topic: %.*s\n", (int) topic.len, topic.data
    );
}
```

Example publish/subscribe subscriber (C)

The following example recipe allows the component to subscribe to all topics.

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.PubSubSubscriberC",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that subscribes to messages.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
          "com.example.PubSubSubscriberC:pubsub:1": {
            "policyDescription": "Allows access to subscribe to all topics.",
            "operations": ["aws.greengrass#SubscribeToTopic"],
            "resources": ["*"]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux",
        "runtime": "*"
      },
      "Lifecycle": {
        "run": "{artifacts:path}/sample_subscribe_to_topic"
      },
      "Artifacts": [
        {
```

```
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.PubSubSubscriberC/1.0.0/sample_subscribe_to_topic",
          "Permission": {
            "Execute": "OWNER"
          }
        }
      ]
    }
  ]
}
```

The following example C application demonstrates how to use the publish/subscribe IPC service to subscribe to messages from other components.

```
#include <assert.h>
#include <gg/error.h>
#include <gg/ipc/client.h>
#include <gg/object.h>
#include <gg/sdk.h>
#include <gg/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

static void on_subscription_response(
    void *ctx, GgBuffer topic, GgObject payload, GgIpcSubscriptionHandle handle
) {
    (void) ctx;
    (void) handle;

    if (gg_obj_type(payload) == GG_TYPE_BUF) {
        GgBuffer message = gg_obj_into_buf(payload);
        printf(
            "Received new message on topic %.*s: %.*s\n",
            (int) topic.len,
            topic.data,
            (int) message.len,
            message.data
        );
    } else {
        assert(gg_obj_type(payload) == GG_TYPE_MAP);
        printf(
            "Received new message on topic %.*s: (JSON message)\n",
```

```
            (int) topic.len,
            topic.data
        );
    }
}

int main(void) {
    gg_sdk_init();

    GgError err = ggipc_connect();
    if (err != GG_ERR_OK) {
        fprintf(stderr, "Failed to establish IPC connection.\n");
        exit(-1);
    }

    GgBuffer topic = GG_STR("my/topic");

    GgIpcSubscriptionHandle handle;
    err = ggipc_subscribe_to_topic(
        topic, on_subscription_response, NULL, &handle
    );
    if (err != GG_ERR_OK) {
        fprintf(
            stderr,
            "Failed to subscribe to topic: %.*s\n",
            (int) topic.len,
            topic.data
        );
        exit(-1);
    }

    printf(
        "Successfully subscribed to topic: %.*s\n", (int) topic.len, topic.data
    );

    // Keep the main thread alive, or the process will exit.
    while (1) {
        sleep(10);
    }

    // To stop subscribing, close the stream.
    ggipc_close_subscription(handle);
}
```

Example publish/subscribe publisher (C++, Component SDK)

The following example recipe allows the component to publish to all topics.

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.PubSubPublisherCpp",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that publishes messages.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
          "com.example.PubSubPublisherCpp:pubsub:1": {
            "policyDescription": "Allows access to publish to all topics.",
            "operations": ["aws.greengrass#PublishToTopic"],
            "resources": ["*"]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux",
        "runtime": "*"
      },
      "Lifecycle": {
        "run": "{artifacts:path}/sample_cpp_publish_to_topic"
      },
      "Artifacts": [
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.PubSubPublisherCpp/1.0.0/sample_cpp_publish_to_topic",
          "Permission": {
            "Execute": "OWNER"
          }
        }
      ]
    }
  ]
```

```
}
```

The following example C++ application demonstrates how to use the publish/subscribe IPC service to publish messages to other components.

```
#include <gg/ipc/client.hpp>
#include <iostream>

int main() {
    auto &client = gg::ipc::Client::get();

    auto error = client.connect();
    if (error) {
        std::cerr << "Failed to establish IPC connection.\n";
        exit(-1);
    }

    std::string_view message = "Hello, World";
    std::string_view topic = "my/topic";

    error = client.publish_to_topic(topic, message);
    if (error) {
        std::cerr << "Failed to publish to topic: " << topic << "\n";
        exit(-1);
    }

    std::cout << "Successfully published to topic: " << topic << "\n";
}
```

Example publish/subscribe subscriber (C++, Component SDK)

The following example recipe allows the component to subscribe to all topics.

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.PubSubSubscriberCpp",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that subscribes to messages.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.pubsub": {
```

```
          "com.example.PubSubSubscriberCpp:pubsub:1": {
            "policyDescription": "Allows access to subscribe to all topics.",
            "operations": ["aws.greengrass#SubscribeToTopic"],
            "resources": ["*"]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux",
        "runtime": "*"
      },
      "Lifecycle": {
        "run": "{artifacts:path}/sample_cpp_subscribe_to_topic"
      },
      "Artifacts": [
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.PubSubSubscriberCpp/1.0.0/sample_cpp_subscribe_to_topic",
          "Permission": {
            "Execute": "OWNER"
          }
        }
      ]
    }
  ]
}
```

The following example C++ application demonstrates how to use the publish/subscribe IPC service to subscribe to messages from other components.

```
#include <gg/ipc/client.hpp>
#include <gg/object.hpp>
#include <unistd.h>
#include <cassert>
#include <iostream>

class ResponseHandler : public gg::ipc::LocalTopicCallback {
    void operator()(
        std::string_view topic,
```

```
        gg::Object payload,
        gg::ipc::Subscription &handle
    ) override {
        (void) handle;
        if (payload.index() == GG_TYPE_BUF) {
            std::cout << "Received new message on topic " << topic << ": "
                      << get<gg::Buffer>(payload) << "\n";
        } else {
            assert(payload.index() == GG_TYPE_MAP);
            std::cout << "Received new message on topic " << topic
                      << ": (JSON message)\n";
        }
    }
};

int main() {
    auto &client = gg::ipc::Client::get();

    auto error = client.connect();
    if (error) {
        std::cerr << "Failed to establish IPC connection.\n";
        exit(-1);
    }

    std::string_view topic = "my/topic";

    static ResponseHandler handler;
    error = client.subscribe_to_topic(topic, handler);
    if (error) {
        std::cerr << "Failed to subscribe to topic: " << topic << "\n";
        exit(-1);
    }

    std::cout << "Successfully subscribed to topic: " << topic << "\n";

    // Keep the main thread alive, or the process will exit.
    while (1) {
        sleep(10);
    }
}
```

Publish/subscribe AWS IoT Core MQTT messages

The AWS IoT Core MQTT messaging IPC service lets you send and receive MQTT messages to and from AWS IoT Core. Components can publish messages to AWS IoT Core and subscribe to topics to act on MQTT messages from other sources. For more information about the AWS IoT Core implementation of MQTT, see MQTT in the *AWS IoT Core Developer Guide*.

> **Note**
>
> This MQTT messaging IPC service lets you exchange messages with AWS IoT Core. For more information about how to exchange messages between components, see Publish/subscribe local messages.

Topics

- Minimum SDK versions
- Authorization
- PublishToIoTCore
- SubscribeToIoTCore
- SubscribeToIoTCoreConnectionStatus
- Examples

Minimum SDK versions

The following table lists the minimum versions of the SDKs that you can use to publish and subscribe to MQTT messages to and from AWS IoT Core.

SDK	Minimum version	
AWS IoT Greengrass Component SDK (C, C++, Rust)	v1.0.0	
AWS IoT Device SDK for Java v2	v1.2.10	

SDK	Minimum version	
AWS IoT Device SDK for Python v2	v1.5.3	
AWS IoT Device SDK for C++ v2	v1.17.0	
AWS IoT Device SDK for JavaScript v2	v1.12.0	

Authorization

To use AWS IoT Core MQTT messaging in a custom component, you must define authorization policies that allow your component to send and receive messages on topics. For information about defining authorization policies, see Authorize components to perform IPC operations.

Authorization policies for AWS IoT Core MQTT messaging have the following properties.

IPC service identifier: `aws.greengrass.ipc.mqttproxy`

Operation	Description	Resources
`aws.greengrass#PublishToIoTCore`	Allows a component to publish messages to AWS IoT Core on the MQTT topics that you specify.	A topic string, such as `test/topic` , or * to allow access to all topics. You can use MQTT topic wildcards (# and +) to match multiple resources.
`aws.greengrass#SubscribeToIoTCore`	Allows a component to subscribe to messages from AWS IoT Core on the topics that you specify.	A topic string, such as `test/topic` , or * to allow access to all topics. You can use MQTT topic wildcards (# and +) to match multiple resources.

Operation	Description	Resources
*	Allows a component to publish and subscribe to AWS IoT Core MQTT messages for the topics that you specify.	A topic string, such as `test/topic` , or * to allow access to all topics. You can use MQTT topic wildcards (# and +) to match multiple resources.

MQTT wildcards in AWS IoT Core MQTT authorization policies

You can use MQTT wildcards in AWS IoT Core MQTT IPC authorization policies. Components can publish and subscribe to topics that match the topic filter that you allow in an authorization policy. For example, if a component's authorization policy grants access to `test/topic/#`, the component can subscribe to `test/topic/#`, and it can publish and subscribe to `test/topic/filter`.

Recipe variables in AWS IoT Core MQTT authorization policies

If you use v2.6.0 or later of the Greengrass nucleus, you can use the `{iot:thingName}` recipe variable in authorization policies. This feature enables you to configure a single authorization policy for a group of core devices, where each core device can access only topics that contain its own name. For example, you can allow a component access to the following topic resource.

```
devices/{iot:thingName}/messages
```

For more information, see Recipe variables and Use recipe variables in merge updates.

Authorization policy examples

You can reference the following authorization policy examples to help you configure authorization policies for your components.

Example authorization policy with unrestricted access

The following example authorization policy allows a component to publish and subscribe to all topics.

JSON

```
{
  "accessControl": {
    "aws.greengrass.ipc.mqttproxy": {
      "com.example.MyIoTCorePubSubComponent:mqttproxy:1": {
        "policyDescription": "Allows access to publish/subscribe to all topics.",
        "operations": [
          "aws.greengrass#PublishToIoTCore",
          "aws.greengrass#SubscribeToIoTCore"
        ],
        "resources": [
          "*"
        ]
      }
    }
  }
}
```

YAML

```
---
accessControl:
  aws.greengrass.ipc.mqttproxy:
    com.example.MyIoTCorePubSubComponent:mqttproxy:1:
      policyDescription: Allows access to publish/subscribe to all topics.
      operations:
        - aws.greengrass#PublishToIoTCore
        - aws.greengrass#SubscribeToIoTCore
      resources:
        - "*"
```

Example authorization policy with limited access

The following example authorization policy allows a component to publish and subscribe to two topics named `factory/1/events` and `factory/1/actions`.

JSON

```
{
  "accessControl": {
    "aws.greengrass.ipc.mqttproxy": {
```

```
      "com.example.MyIoTCorePubSubComponent:mqttproxy:1": {
        "policyDescription": "Allows access to publish/subscribe to factory 1
 topics.",
        "operations": [
          "aws.greengrass#PublishToIoTCore",
          "aws.greengrass#SubscribeToIoTCore"
        ],
        "resources": [
          "factory/1/actions",
          "factory/1/events"
        ]
      }
    }
  }
}
```

YAML

```
---
accessControl:
  aws.greengrass.ipc.mqttproxy:
    "com.example.MyIoTCorePubSubComponent:mqttproxy:1":
      policyDescription: Allows access to publish/subscribe to factory 1 topics.
      operations:
        - aws.greengrass#PublishToIoTCore
        - aws.greengrass#SubscribeToIoTCore
      resources:
        - factory/1/actions
        - factory/1/events
```

Example authorization policy for a group of core devices

> **Important**
>
> This example uses a feature that is available for v2.6.0 and later of the Greengrass nucleus component. Greengrass nucleus v2.6.0 adds support for most recipe variables, such as `{iot:thingName}`, in component configurations.

The following example authorization policy allows a component to publish and subscribe to a topic that contains the name of the core device that runs the component.

JSON

```
{
  "accessControl": {
    "aws.greengrass.ipc.mqttproxy": {
      "com.example.MyIoTCorePubSubComponent:mqttproxy:1": {
        "policyDescription": "Allows access to publish/subscribe to all topics.",
        "operations": [
          "aws.greengrass#PublishToIoTCore",
          "aws.greengrass#SubscribeToIoTCore"
        ],
        "resources": [
          "factory/1/devices/{iot:thingName}/controls"
        ]
      }
    }
  }
}
```

YAML

```
---
accessControl:
  aws.greengrass.ipc.mqttproxy:
    "com.example.MyIoTCorePubSubComponent:mqttproxy:1":
      policyDescription: Allows access to publish/subscribe to all topics.
      operations:
        - aws.greengrass#PublishToIoTCore
        - aws.greengrass#SubscribeToIoTCore
      resources:
        - factory/1/devices/{iot:thingName}/controls
```

PublishToIoTCore

Publishes an MQTT message to AWS IoT Core on a topic.

When you publish MQTT messages to AWS IoT Core, there is a quota of 100 transactions per second. If you exceed this quota, messages are queued for processing on the Greengrass device. There is also a quota of 512 Kb of data per second and an account-wide quota of 20,000 publishes per second (2,000 in some AWS Regions). For more information about MQTT message broker limits in AWS IoT Core, see AWS IoT Core message broker and protocol limits and quotas.

If you exceed these quotas, the Greengrass device limits publishing messages to AWS IoT Core. Messages are stored in a spooler in memory. By default, the memory allocated to the spooler is 2.5 Mb. If the spooler fills up, new messages are rejected. You can increase the size of the spooler. For more information, see Configuration in the Greengrass nucleus documentation. To avoid filling the spooler and needing to increase the allocated memory, limit publish requests to no more than 100 requests per second.

When your application needs to send messages at a higher rate, or larger messages, consider using the Stream manager to send messages to Kinesis Data Streams. The stream manager component is designed to transfer high-volume data to the AWS Cloud. For more information, see Manage data streams on Greengrass core devices.

Request

This operation's request has the following parameters:

`topicName` (Python: `topic_name`)

The topic to which to publish the message.

`qos`

The MQTT QoS to use. This enum, `QOS`, has the following values:

- `AT_MOST_ONCE` – QoS 0. The MQTT message is delivered at most once.
- `AT_LEAST_ONCE` – QoS 1. The MQTT message is delivered at least once.

`payload`

(Optional) The message payload as a blob.

The following features are available for v2.10.0 and later of the Greengrass nucleus when using MQTT 5. These features are ignored when you are using MQTT 3.1.1. The following table lists the minimum version of the AWS IoT device SDK that you must use to access these features.

SDK	Minimum version
AWS IoT Device SDK for Python v2	v1.15.0
AWS IoT Device SDK for Java v2	v1.13.0

SDK	Minimum version
AWS IoT Device SDK for C++ v2	v1.24.0
AWS IoT Device SDK for JavaScript v2	v1.13.0

`payloadFormat`

(Optional) The format of the message payload. If you don't set the `payloadFormat`, the type is assumed to be BYTES. The enum has the following values:

- BYTES – The content of the payload is a binary blob.
- UTF8 – The content of the payload is a UTF8 string of characters.

`retain`

(Optional) Indicates whether to set the MQTT retain option to `true` when publishing.

`userProperties`

(Optional) A list of application-specific `UserProperty` objects to send. The `UserProperty` object is defined as follows:

```
UserProperty:
  key: string
  value: string
```

`messageExpiryIntervalSeconds`

(Optional) The number of seconds before the message expires and is deleted by the server. If this value is not set, the message doesn't expire.

`correlationData`

(Optional) Information added to the request that can be used to associate a request with a response.

`responseTopic`

(Optional) The topic that should be used for the response message.

`contentType`

(Optional) An application-specific identifier of the content type of the message.

Response

This operation doesn't provide any information in its response.

Examples

The following examples demonstrate how to call this operation in custom component code.

Java (IPC client V2)

Example: Publish a message

```
package com.aws.greengrass.docs.samples.ipc;

import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClientV2;
import software.amazon.awssdk.aws.greengrass.model.PublishToIoTCoreRequest;
import software.amazon.awssdk.aws.greengrass.model.QOS;
import java.nio.charset.StandardCharsets;

public class PublishToIoTCore {

    public static void main(String[] args) {
        String topic = args[0];
        String message = args[1];
        QOS qos = QOS.get(args[2]);

        try (GreengrassCoreIPCClientV2 ipcClientV2 =
 GreengrassCoreIPCClientV2.builder().build()) {
            ipcClientV2.publishToIoTCore(new PublishToIoTCoreRequest()
                    .withTopicName(topic)
                    .withPayload(message.getBytes(StandardCharsets.UTF_8))
                    .withQos(qos));
            System.out.println("Successfully published to topic: " + topic);
        } catch (Exception e) {
            System.err.println("Exception occurred.");
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

Python (IPC client V2)

Example: Publish a message

> **Note**
>
> This example assumes that you are using version 1.5.4 or later of the AWS IoT Device SDK for Python v2.

```
import awsiot.greengrasscoreipc.clientv2 as clientV2

topic = 'my/topic'
qos = '1'
payload = 'Hello, World'

ipc_client = clientV2.GreengrassCoreIPCClientV2()
resp = ipc_client.publish_to_iot_core(topic_name=topic, qos=qos, payload=payload)
ipc_client.close()
```

Java (IPC client V1)

Example: Publish a message

> **Note**
>
> This example uses an `IPCUtils` class to create a connection to the AWS IoT Greengrass Core IPC service. For more information, see Connect to the AWS IoT Greengrass Core IPC service.

```
package com.aws.greengrass.docs.samples.ipc;

import com.aws.greengrass.docs.samples.ipc.util.IPCUtils;
import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClient;
import software.amazon.awssdk.aws.greengrass.PublishToIoTCoreResponseHandler;
import software.amazon.awssdk.aws.greengrass.model.PublishToIoTCoreRequest;
import software.amazon.awssdk.aws.greengrass.model.PublishToIoTCoreResponse;
import software.amazon.awssdk.aws.greengrass.model.QOS;
import software.amazon.awssdk.aws.greengrass.model.UnauthorizedError;
import software.amazon.awssdk.eventstreamrpc.EventStreamRPCConnection;
```

```
import java.nio.charset.StandardCharsets;
import java.util.Optional;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public class PublishToIoTCore {

    public static final int TIMEOUT_SECONDS = 10;

    public static void main(String[] args) {
        String topic = args[0];
        String message = args[1];
        QOS qos = QOS.get(args[2]);
        try (EventStreamRPCConnection eventStreamRPCConnection =
                     IPCUtils.getEventStreamRpcConnection()) {
            GreengrassCoreIPCClient ipcClient =
                    new GreengrassCoreIPCClient(eventStreamRPCConnection);
            PublishToIoTCoreResponseHandler responseHandler =
                    PublishToIoTCore.publishBinaryMessageToTopic(ipcClient, topic,
 message, qos);
            CompletableFuture<PublishToIoTCoreResponse> futureResponse =
                    responseHandler.getResponse();
            try {
                futureResponse.get(TIMEOUT_SECONDS, TimeUnit.SECONDS);
                System.out.println("Successfully published to topic: " + topic);
            } catch (TimeoutException e) {
                System.err.println("Timeout occurred while publishing to topic: " +
 topic);
            } catch (ExecutionException e) {
                if (e.getCause() instanceof UnauthorizedError) {
                    System.err.println("Unauthorized error while publishing to
 topic: " + topic);
                } else {
                    throw e;
                }
            }
        } catch (InterruptedException e) {
            System.out.println("IPC interrupted.");
        } catch (ExecutionException e) {
            System.err.println("Exception occurred when using IPC.");
            e.printStackTrace();
```

```
            System.exit(1);
        }
    }

    public static PublishToIoTCoreResponseHandler
 publishBinaryMessageToTopic(GreengrassCoreIPCClient greengrassCoreIPCClient, String
 topic, String message, QOS qos) {
        PublishToIoTCoreRequest publishToIoTCoreRequest = new
 PublishToIoTCoreRequest();
        publishToIoTCoreRequest.setTopicName(topic);

 publishToIoTCoreRequest.setPayload(message.getBytes(StandardCharsets.UTF_8));
        publishToIoTCoreRequest.setQos(qos);
        return greengrassCoreIPCClient.publishToIoTCore(publishToIoTCoreRequest,
 Optional.empty());
    }
}
```

Python (IPC client V1)

Example: Publish a message

> **Note**
>
> This example assumes that you are using version 1.5.4 or later of the AWS IoT Device SDK for Python v2.

```
import awsiot.greengrasscoreipc
import awsiot.greengrasscoreipc.client as client
from awsiot.greengrasscoreipc.model import (
    QOS,
    PublishToIoTCoreRequest
)

TIMEOUT = 10

ipc_client = awsiot.greengrasscoreipc.connect()

topic = "my/topic"
message = "Hello, World"
qos = QOS.AT_LEAST_ONCE
```

```
request = PublishToIoTCoreRequest()
request.topic_name = topic
request.payload = bytes(message, "utf-8")
request.qos = qos
operation = ipc_client.new_publish_to_iot_core()
operation.activate(request)
future_response = operation.get_response()
future_response.result(TIMEOUT)
```

C++ (IPC client V1)

Example: Publish a message

```
#include <iostream>

#include <aws/crt/Api.h>
#include <aws/greengrass/GreengrassCoreIpcClient.h>

using namespace Aws::Crt;
using namespace Aws::Greengrass;

class IpcClientLifecycleHandler : public ConnectionLifecycleHandler {
    void OnConnectCallback() override {
        // Handle connection to IPC service.
    }

    void OnDisconnectCallback(RpcError error) override {
        // Handle disconnection from IPC service.
    }

    bool OnErrorCallback(RpcError error) override {
        // Handle IPC service connection error.
        return true;
    }
};

int main() {
    ApiHandle apiHandle(g_allocator);
    Io::EventLoopGroup eventLoopGroup(1);
    Io::DefaultHostResolver socketResolver(eventLoopGroup, 64, 30);
    Io::ClientBootstrap bootstrap(eventLoopGroup, socketResolver);
    IpcClientLifecycleHandler ipcLifecycleHandler;
    GreengrassCoreIpcClient ipcClient(bootstrap);
    auto connectionStatus = ipcClient.Connect(ipcLifecycleHandler).get();
```

```
    if (!connectionStatus) {
        std::cerr << "Failed to establish IPC connection: " <<
connectionStatus.StatusToString() << std::endl;
        exit(-1);
    }

    String message("Hello, World!");
    String topic("my/topic");
    QOS qos = QOS_AT_MOST_ONCE;
    int timeout = 10;

    PublishToIoTCoreRequest request;
    Vector<uint8_t> messageData({message.begin(), message.end()});
    request.SetTopicName(topic);
    request.SetPayload(messageData);
    request.SetQos(qos);

    auto operation = ipcClient.NewPublishToIoTCore();
    auto activate = operation->Activate(request, nullptr);
    activate.wait();

    auto responseFuture = operation->GetResult();
    if (responseFuture.wait_for(std::chrono::seconds(timeout)) ==
std::future_status::timeout) {
        std::cerr << "Operation timed out while waiting for response from Greengrass
Core." << std::endl;
        exit(-1);
    }

    auto response = responseFuture.get();
    if (!response) {
        // Handle error.
        auto errorType = response.GetResultType();
        if (errorType == OPERATION_ERROR) {
            auto *error = response.GetOperationError();
            (void)error;
            // Handle operation error.
        } else {
            // Handle RPC error.
        }
    }

    return 0;
```

```
}
```

JavaScript

Example: Publish a message

```
import * as greengrasscoreipc from "aws-iot-device-sdk-v2/dist/greengrasscoreipc";
import {QOS, PublishToIoTCoreRequest} from "aws-iot-device-sdk-v2/dist/
greengrasscoreipc/model";

class PublishToIoTCore {
    private ipcClient: greengrasscoreipc.Client
    private readonly topic: string;

    constructor() {
        // define your own constructor, e.g.
        this.topic = "<define_your_topic>";
        this.publishToIoTCore().then(r => console.log("Started workflow"));
    }

    private async publishToIoTCore() {
        try {
            const request: PublishToIoTCoreRequest = {
                topicName: this.topic,
                qos: QOS.AT_LEAST_ONCE, // you can change this depending on your use
 case
            }

            this.ipcClient = await getIpcClient();

            await this.ipcClient.publishToIoTCore(request);
        } catch (e) {
            // parse the error depending on your use cases
            throw e
        }
    }
}


export async function getIpcClient(){
    try {
        const ipcClient = greengrasscoreipc.createClient();
```

```
        await ipcClient.connect()
            .catch(error => {
                // parse the error depending on your use cases
                throw error;
            });
        return ipcClient
    } catch (err) {
        // parse the error depending on your use cases
        throw err
    }
}

// starting point
const publishToIoTCore = new PublishToIoTCore();
```

Rust

Example: Publish a message

```
use gg_sdk::{Qos, Sdk};

fn main() {
    let sdk = Sdk::init();
    sdk.connect().expect("Failed to establish IPC connection");

    let message = b"Hello, World";
    let topic = "my/topic";
    let qos = Qos::AtLeastOnce;

    sdk.publish_to_iot_core(topic, message, qos)
        .expect("Failed to publish to topic");

    println!("Successfully published to topic: {topic}");
}
```

C

Example: Publish a message

```
#include <gg/error.h>
#include <gg/ipc/client.h>
#include <gg/sdk.h>
#include <stdio.h>
```

```
#include <stdlib.h>

int main(void) {
    gg_sdk_init();

    GgError err = ggipc_connect();
    if (err != GG_ERR_OK) {
        fprintf(stderr, "Failed to establish IPC connection.\n");
        exit(-1);
    }

    GgBuffer message = GG_STR("Hello, World");
    GgBuffer topic = GG_STR("my/topic");
    uint8_t qos = 1;

    err = ggipc_publish_to_iot_core(topic, message, qos);
    if (err != GG_ERR_OK) {
        fprintf(
            stderr,
            "Failed to publish to topic: %.*s\n",
            (int) topic.len,
            topic.data
        );
        exit(-1);
    }

    printf(
        "Successfully published to topic: %.*s\n", (int) topic.len, topic.data
    );
}
```

C++ (Component SDK)

Example: Publish a message

```
#include <gg/ipc/client.hpp>
#include <iostream>

int main() {
    auto &client = gg::ipc::Client::get();

    auto error = client.connect();
    if (error) {
        std::cerr << "Failed to establish IPC connection.\n";
```

```
        exit(-1);
    }

    std::string_view message = "Hello, World";
    std::string_view topic = "my/topic";
    uint8_t qos = 1;

    error = client.publish_to_iot_core(topic, message, qos);
    if (error) {
        std::cerr << "Failed to publish to topic: " << topic << "\n";
        exit(-1);
    }

    std::cout << "Successfully published to topic: " << topic << "\n";
}
```

SubscribeToIoTCore

Subscribe to MQTT messages from AWS IoT Core on a topic or topic filter. The AWS IoT Greengrass Core software removes subscriptions when the component reaches the end of its lifecycle.

This operation is a subscription operation where you subscribe to a stream of event messages. To use this operation, define a stream response handler with functions that handle event messages, errors, and stream closure. For more information, see Subscribe to IPC event streams.

Event message type: `IoTCoreMessage`

Request

This operation's request has the following parameters:

`topicName` (Python: `topic_name`)

The topic to which to subscribe. You can use MQTT topic wildcards (# and +) to subscribe to multiple topics.

`qos`

The MQTT QoS to use. This enum, `QOS`, has the following values:

- `AT_MOST_ONCE` – QoS 0. The MQTT message is delivered at most once.
- `AT_LEAST_ONCE` – QoS 1. The MQTT message is delivered at least once.

Response

This operation's response has the following information:

`messages`

The stream of MQTT messages. This object, `IoTCoreMessage`, contains the following information:

`message`

The MQTT message. This object, `MQTTMessage`, contains the following information:

`topicName` (Python: `topic_name`)

The topic to which the message was published.

`payload`

(Optional) The message payload as a blob.

The following features are available for v2.10.0 and later of the Greengrass nucleus when using MQTT 5. These features are ignored when you are using MQTT 3.1.1. The following table lists the minimum version of the AWS IoT device SDK that you must use to access these features.

SDK	Minimum version
AWS IoT Device SDK for Python v2	v1.15.0
AWS IoT Device SDK for Java v2	v1.13.0
AWS IoT Device SDK for C++ v2	v1.24.0
AWS IoT Device SDK for JavaScript v2	v1.13.0

`payloadFormat`

(Optional) The format of the message payload. If you don't set the `payloadFormat`, the type is assumed to be `BYTES`. The enum has the following values:

- `BYTES` – The content of the payload is a binary blob.
- `UTF8` – The content of the payload is a UTF8 string of characters.

`retain`

(Optional) Indicates whether to set the MQTT retain option to `true` when publishing.

`userProperties`

(Optional) A list of application-specific `UserProperty` objects to send. The `UserProperty` object is defined as follows:

```
UserProperty:
  key: string
  value: string
```

`messageExpiryIntervalSeconds`

(Optional) The number of seconds before the message expires and is deleted by the server. If this value is not set, the message doesn't expire.

`correlationData`

(Optional) Information added to the request that can be used to associate a request with a response.

`responseTopic`

(Optional) The topic that should be used for the response message.

`contentType`

(Optional) An application specific identifier of the content type of the message.

Examples

The following examples demonstrate how to call this operation in custom component code.

Java (IPC client V2)

Example: Subscribe to messages

```
package com.aws.greengrass.docs.samples.ipc;

import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClientV2;
import software.amazon.awssdk.aws.greengrass.SubscribeToIoTCoreResponseHandler;
import software.amazon.awssdk.aws.greengrass.model.QOS;
import software.amazon.awssdk.aws.greengrass.model.IoTCoreMessage;
```

```
import software.amazon.awssdk.aws.greengrass.model.SubscribeToIoTCoreRequest;
import software.amazon.awssdk.aws.greengrass.model.SubscribeToIoTCoreResponse;

import java.nio.charset.StandardCharsets;
import java.util.Optional;
import java.util.function.Consumer;
import java.util.function.Function;


public class SubscribeToIoTCore {

    public static void main(String[] args) {
        String topic = args[0];
        QOS qos = QOS.get(args[1]);

        Consumer<IoTCoreMessage> onStreamEvent = ioTCoreMessage ->
                System.out.printf("Received new message on topic %s: %s%n",
                        ioTCoreMessage.getMessage().getTopicName(),
                        new String(ioTCoreMessage.getMessage().getPayload(),
 StandardCharsets.UTF_8));

        Optional<Function<Throwable, Boolean>> onStreamError =
                Optional.of(e -> {
                    System.err.println("Received a stream error.");
                    e.printStackTrace();
                    return false;
                });

        Optional<Runnable> onStreamClosed = Optional.of(() ->
                System.out.println("Subscribe to IoT Core stream closed."));

        try (GreengrassCoreIPCClientV2 ipcClientV2 =
 GreengrassCoreIPCClientV2.builder().build()) {
            SubscribeToIoTCoreRequest request = new SubscribeToIoTCoreRequest()
                    .withTopicName(topic)
                    .withQos(qos);

            GreengrassCoreIPCClientV2.StreamingResponse<SubscribeToIoTCoreResponse,
 SubscribeToIoTCoreResponseHandler>
                    streamingResponse = ipcClientV2.subscribeToIoTCore(request,
 onStreamEvent, onStreamError, onStreamClosed);

            streamingResponse.getResponse();
            System.out.println("Successfully subscribed to topic: " + topic);
```

```
            // Keep the main thread alive, or the process will exit.
            while (true) {
                Thread.sleep(10000);
            }

            // To stop subscribing, close the stream.
            streamingResponse.getHandler().closeStream();
        } catch (InterruptedException e) {
            System.out.println("Subscribe interrupted.");
        } catch (Exception e) {
            System.err.println("Exception occurred.");
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

Python (IPC client V2)

Example: subscribe to messages

> **Note**
>
> This example assumes that you are using version 1.5.4 or later of the AWS IoT Device SDK for Python v2.

```
import threading
import traceback

import awsiot.greengrasscoreipc.clientv2 as clientV2

topic = 'my/topic'
qos = '1'

def on_stream_event(event):
    try:
        topic_name = event.message.topic_name
        message = str(event.message.payload, 'utf-8')
        print(f'Received new message on topic {topic_name}:  {message}')
    except:
```

```
        traceback.print_exc()

def on_stream_error(error):
    # Return True to close stream, False to keep stream open.
    return True

def on_stream_closed():
    pass

ipc_client = clientV2.GreengrassCoreIPCClientV2()
resp, operation = ipc_client.subscribe_to_iot_core(
    topic_name=topic,
    qos=qos,
    on_stream_event=on_stream_event,
    on_stream_error=on_stream_error,
    on_stream_closed=on_stream_closed
)

# Keep the main thread alive, or the process will exit.
event = threading.Event()
event.wait()

# To stop subscribing, close the operation stream.
operation.close()
ipc_client.close()
```

Java (IPC client V1)

Example: Subscribe to messages

> **Note**
>
> This example uses an `IPCUtils` class to create a connection to the AWS IoT Greengrass Core IPC service. For more information, see Connect to the AWS IoT Greengrass Core IPC service.

```
package com.aws.greengrass.docs.samples.ipc;

import com.aws.greengrass.docs.samples.ipc.util.IPCUtils;
import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClient;
import software.amazon.awssdk.aws.greengrass.SubscribeToIoTCoreResponseHandler;
```

```
import software.amazon.awssdk.aws.greengrass.model.*;
import software.amazon.awssdk.eventstreamrpc.EventStreamRPCConnection;
import software.amazon.awssdk.eventstreamrpc.StreamResponseHandler;

import java.nio.charset.StandardCharsets;
import java.util.Optional;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public class SubscribeToIoTCore {

    public static final int TIMEOUT_SECONDS = 10;

    public static void main(String[] args) {
        String topic = args[0];
        QOS qos = QOS.get(args[1]);
        try (EventStreamRPCConnection eventStreamRPCConnection =
                     IPCUtils.getEventStreamRpcConnection()) {
            GreengrassCoreIPCClient ipcClient =
                    new GreengrassCoreIPCClient(eventStreamRPCConnection);
            StreamResponseHandler<IoTCoreMessage> streamResponseHandler =
                    new SubscriptionResponseHandler();
            SubscribeToIoTCoreResponseHandler responseHandler =
                    SubscribeToIoTCore.subscribeToIoTCore(ipcClient, topic, qos,
                            streamResponseHandler);
            CompletableFuture<SubscribeToIoTCoreResponse> futureResponse =
                    responseHandler.getResponse();
            try {
                futureResponse.get(TIMEOUT_SECONDS, TimeUnit.SECONDS);
                System.out.println("Successfully subscribed to topic: " + topic);
            } catch (TimeoutException e) {
                System.err.println("Timeout occurred while subscribing to topic: " +
 topic);
            } catch (ExecutionException e) {
                if (e.getCause() instanceof UnauthorizedError) {
                    System.err.println("Unauthorized error while subscribing to
 topic: " + topic);
                } else {
                    throw e;
                }
            }
```

```
            // Keep the main thread alive, or the process will exit.
            try {
                while (true) {
                    Thread.sleep(10000);
                }
            } catch (InterruptedException e) {
                System.out.println("Subscribe interrupted.");
            }

            // To stop subscribing, close the stream.
            responseHandler.closeStream();
        } catch (InterruptedException e) {
            System.out.println("IPC interrupted.");
        } catch (ExecutionException e) {
            System.err.println("Exception occurred when using IPC.");
            e.printStackTrace();
            System.exit(1);
        }
    }

    public static SubscribeToIoTCoreResponseHandler
subscribeToIoTCore(GreengrassCoreIPCClient greengrassCoreIPCClient, String topic,
QOS qos, StreamResponseHandler<IoTCoreMessage> streamResponseHandler) {
        SubscribeToIoTCoreRequest subscribeToIoTCoreRequest = new
SubscribeToIoTCoreRequest();
        subscribeToIoTCoreRequest.setTopicName(topic);
        subscribeToIoTCoreRequest.setQos(qos);
        return
greengrassCoreIPCClient.subscribeToIoTCore(subscribeToIoTCoreRequest,
                Optional.of(streamResponseHandler));
    }

    public static class SubscriptionResponseHandler implements
StreamResponseHandler<IoTCoreMessage> {

        @Override
        public void onStreamEvent(IoTCoreMessage ioTCoreMessage) {
            try {
                String topic = ioTCoreMessage.getMessage().getTopicName();
                String message = new
String(ioTCoreMessage.getMessage().getPayload(),
                        StandardCharsets.UTF_8);
                System.out.printf("Received new message on topic %s: %s%n", topic,
message);
```

```
            } catch (Exception e) {
                System.err.println("Exception occurred while processing subscription
 response " +
                        "message.");
                e.printStackTrace();
            }
        }

        @Override
        public boolean onStreamError(Throwable error) {
            System.err.println("Received a stream error.");
            error.printStackTrace();
            return false;
        }

        @Override
        public void onStreamClosed() {
            System.out.println("Subscribe to IoT Core stream closed.");
        }
    }
}
```

Python (IPC client V1)

Example: Subscribe to messages

> **Note**
>
> This example assumes that you are using version 1.5.4 or later of the AWS IoT Device SDK for Python v2.

```
import time
import traceback

import awsiot.greengrasscoreipc
import awsiot.greengrasscoreipc.client as client
from awsiot.greengrasscoreipc.model import (
    IoTCoreMessage,
    QOS,
    SubscribeToIoTCoreRequest
)
```

```
TIMEOUT = 10

ipc_client = awsiot.greengrasscoreipc.connect()

class StreamHandler(client.SubscribeToIoTCoreStreamHandler):
    def __init__(self):
        super().__init__()

    def on_stream_event(self, event: IoTCoreMessage) -> None:
        try:
            message = str(event.message.payload, "utf-8")
            topic_name = event.message.topic_name
            # Handle message.
        except:
            traceback.print_exc()

    def on_stream_error(self, error: Exception) -> bool:
        # Handle error.
        return True  # Return True to close stream, False to keep stream open.

    def on_stream_closed(self) -> None:
        # Handle close.
        pass


topic = "my/topic"
qos = QOS.AT_MOST_ONCE

request = SubscribeToIoTCoreRequest()
request.topic_name = topic
request.qos = qos
handler = StreamHandler()
operation = ipc_client.new_subscribe_to_iot_core(handler)
operation.activate(request)
future_response = operation.get_response()
future_response.result(TIMEOUT)

# Keep the main thread alive, or the process will exit.
while True:
    time.sleep(10)

# To stop subscribing, close the operation stream.
operation.close()
```

C++ (IPC client V1)

Example: Subscribe to messages

```
#include <iostream>

#include <aws/crt/Api.h>
#include <aws/greengrass/GreengrassCoreIpcClient.h>

using namespace Aws::Crt;
using namespace Aws::Greengrass;

class IoTCoreResponseHandler : public SubscribeToIoTCoreStreamHandler {

    public:
        virtual ~IoTCoreResponseHandler() {}

    private:
        void OnStreamEvent(IoTCoreMessage *response) override {
            auto message = response->GetMessage();
            if (message.has_value() && message.value().GetPayload().has_value()) {
                auto messageBytes = message.value().GetPayload().value();
                std::string messageString(messageBytes.begin(), messageBytes.end());
                std::string topicName =
 message.value().GetTopicName().value().c_str();
                // Handle message.
            }
        }

        bool OnStreamError(OperationError *error) override {
            // Handle error.
            return false; // Return true to close stream, false to keep stream open.
        }

        void OnStreamClosed() override {
            // Handle close.
        }
};

class IpcClientLifecycleHandler : public ConnectionLifecycleHandler {
    void OnConnectCallback() override {
        // Handle connection to IPC service.
    }
```

```
    void OnDisconnectCallback(RpcError error) override {
        // Handle disconnection from IPC service.
    }

    bool OnErrorCallback(RpcError error) override {
        // Handle IPC service connection error.
        return true;
    }
};

int main() {
    ApiHandle apiHandle(g_allocator);
    Io::EventLoopGroup eventLoopGroup(1);
    Io::DefaultHostResolver socketResolver(eventLoopGroup, 64, 30);
    Io::ClientBootstrap bootstrap(eventLoopGroup, socketResolver);
    IpcClientLifecycleHandler ipcLifecycleHandler;
    GreengrassCoreIpcClient ipcClient(bootstrap);
    auto connectionStatus = ipcClient.Connect(ipcLifecycleHandler).get();
    if (!connectionStatus) {
        std::cerr << "Failed to establish IPC connection: " <<
 connectionStatus.StatusToString() << std::endl;
        exit(-1);
    }

    String topic("my/topic");
    QOS qos = QOS_AT_MOST_ONCE;
    int timeout = 10;

    SubscribeToIoTCoreRequest request;
    request.SetTopicName(topic);
    request.SetQos(qos);
    auto streamHandler = MakeShared<IoTCoreResponseHandler>(DefaultAllocator());
    auto operation = ipcClient.NewSubscribeToIoTCore(streamHandler);
    auto activate = operation->Activate(request, nullptr);
    activate.wait();

    auto responseFuture = operation->GetResult();
    if (responseFuture.wait_for(std::chrono::seconds(timeout)) ==
 std::future_status::timeout) {
        std::cerr << "Operation timed out while waiting for response from Greengrass
 Core." << std::endl;
        exit(-1);
    }
```

```
    auto response = responseFuture.get();
    if (!response) {
        // Handle error.
        auto errorType = response.GetResultType();
        if (errorType == OPERATION_ERROR) {
            auto *error = response.GetOperationError();
            (void)error;
            // Handle operation error.
        } else {
            // Handle RPC error.
        }
        exit(-1);
    }

    // Keep the main thread alive, or the process will exit.
    while (true) {
        std::this_thread::sleep_for(std::chrono::seconds(10));
    }

    operation->Close();
    return 0;
}
```

JavaScript

Example: Subscribe to messages

```
import * as greengrasscoreipc from "aws-iot-device-sdk-v2/dist/greengrasscoreipc";
import {IoTCoreMessage, QOS, SubscribeToIoTCoreRequest} from "aws-iot-device-sdk-v2/
dist/greengrasscoreipc/model";
import {RpcError} from "aws-iot-device-sdk-v2/dist/eventstream_rpc";

class SubscribeToIoTCore {
    private ipcClient: greengrasscoreipc.Client
    private readonly topic: string;

    constructor() {
        // define your own constructor, e.g.
        this.topic = "<define_your_topic>";
        this.subscribeToIoTCore().then(r => console.log("Started workflow"));
    }

    private async subscribeToIoTCore() {
```

```
        try {
            const request: SubscribeToIoTCoreRequest = {
                topicName: this.topic,
                qos: QOS.AT_LEAST_ONCE, // you can change this depending on your use
 case
            }

            this.ipcClient = await getIpcClient();

            const streamingOperation = this.ipcClient.subscribeToIoTCore(request);

            streamingOperation.on('message', (message: IoTCoreMessage) => {
                // parse the message depending on your use cases, e.g.
                if (message.message && message.message.payload) {
                    const receivedMessage = message.message.payload.toString();
                }
            });

            streamingOperation.on('streamError', (error : RpcError) => {
                // define your own error handling logic
            });

            streamingOperation.on('ended', () => {
                // define your own logic
            });

            await streamingOperation.activate();

            // Keep the main thread alive, or the process will exit.
            await new Promise((resolve) => setTimeout(resolve, 10000))
        } catch (e) {
            // parse the error depending on your use cases
            throw e
        }
    }
}

export async function getIpcClient(){
    try {
        const ipcClient = greengrasscoreipc.createClient();
        await ipcClient.connect()
            .catch(error => {
                // parse the error depending on your use cases
                throw error;
```

```
            });
        return ipcClient
    } catch (err) {
        // parse the error depending on your use cases
        throw err
    }
}

// starting point
const subscribeToIoTCore = new SubscribeToIoTCore();
```

Rust

Example: Subscribe to messages

```
use gg_sdk::{Qos, Sdk};
use std::{thread, time::Duration};

fn main() {
    let sdk = Sdk::init();
    sdk.connect().expect("Failed to establish IPC connection");

    let topic = "my/topic";
    let qos = Qos::AtLeastOnce;

    let callback = |topic: &str, payload: &[u8]| {
        let message = String::from_utf8_lossy(payload);
        println!("Received new message on topic {topic}: {message}");
    };

    let _sub = sdk
        .subscribe_to_iot_core(topic, qos, &callback)
        .expect("Failed to subscribe to topic");

    println!("Successfully subscribed to topic: {topic}");

    // Keep the main thread alive, or the process will exit.
    loop {
        thread::sleep(Duration::from_secs(10));
    }
}
```

C

Example: Subscribe to messages

```
#include <gg/error.h>
#include <gg/ipc/client.h>
#include <gg/sdk.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

static void on_subscription_response(
    void *ctx, GgBuffer topic, GgBuffer payload, GgIpcSubscriptionHandle handle
) {
    (void) ctx;
    (void) handle;

    printf(
        "Received new message on topic %.*s: %.*s\n",
        (int) topic.len,
        topic.data,
        (int) payload.len,
        payload.data
    );
}

int main(void) {
    gg_sdk_init();

    GgError err = ggipc_connect();
    if (err != GG_ERR_OK) {
        fprintf(stderr, "Failed to establish IPC connection.\n");
        exit(-1);
    }

    GgBuffer topic = GG_STR("my/topic");
    uint8_t qos = 1;

    GgIpcSubscriptionHandle handle;
    err = ggipc_subscribe_to_iot_core(
        topic, qos, on_subscription_response, NULL, &handle
    );
    if (err != GG_ERR_OK) {
        fprintf(
```

```
            stderr,
            "Failed to subscribe to topic: %.*s\n",
            (int) topic.len,
            topic.data
        );
        exit(-1);
    }

    printf(
        "Successfully subscribed to topic: %.*s\n", (int) topic.len, topic.data
    );

    // Keep the main thread alive, or the process will exit.
    while (1) {
        sleep(10);
    }

    // To stop subscribing, close the subscription handle.
    ggipc_close_subscription(handle);
}
```

C++ (Component SDK)

Example: Subscribe to messages

```
#include <gg/ipc/client.hpp>
#include <unistd.h>
#include <iostream>

class ResponseHandler : public gg::ipc::IotTopicCallback {
    void operator()(
        std::string_view topic,
        gg::Buffer payload,
        gg::ipc::Subscription &handle
    ) override {
        (void) handle;
        std::cout << "Received new message on topic " << topic << ": "
                  << payload << "\n";
    }
};

int main() {
    auto &client = gg::ipc::Client::get();
```

```
    auto error = client.connect();
    if (error) {
        std::cerr << "Failed to establish IPC connection.\n";
        exit(-1);
    }

    std::string_view topic = "my/topic";
    uint8_t qos = 1;

    static ResponseHandler handler;
    error = client.subscribe_to_iot_core(topic, qos, handler);
    if (error) {
        std::cerr << "Failed to subscribe to topic: " << topic << "\n";
        exit(-1);
    }

    std::cout << "Successfully subscribed to topic: " << topic << "\n";

    // Keep the main thread alive, or the process will exit.
    while (1) {
        sleep(10);
    }
}
```

SubscribeToIoTCoreConnectionStatus

Use this operation to subscribe to the status of the MQTT connection between the core device and AWS IoT Core. After you subscribe, you receive the current connection status as the first event. Then, you receive an event each time the core device connects to or disconnects from AWS IoT Core.

This operation is a subscription operation where you subscribe to a stream of event messages. To use this operation, define a stream response handler with functions that handle event messages, errors, and stream closure. For more information, see Subscribe to IPC event streams.

Event message type: `IoTCoreConnectionStatusEvent`

Note

This operation doesn't require an authorization policy, because it's an informational, local-only operation that doesn't expose MQTT topic data. You can call this operation in your components without an `accessControl` entry in the component recipe.

Request

This operation's request doesn't have any parameters.

Response

This operation's response has the following information:

`connectionStatusEvent` (Python: `connection_status_event`)

The `ConnectionStatusEvent` object in each streamed `IoTCoreConnectionStatusEvent`. This object reports the MQTT connection state of the core device, and contains the following information:

`status`

The status of the MQTT connection between the core device and AWS IoT Core. This enum, `ConnectionStatus`, has the following values:

- `CONNECTED` – The core device is connected to AWS IoT Core.
- `DISCONNECTED` – The core device is disconnected from AWS IoT Core.

Example events

Each event in the stream is the JSON serialization of an `IoTCoreConnectionStatusEvent` object. The `connectionStatusEvent` key identifies the type of event in the stream.

After you subscribe, you receive the following example event when the core device is connected to AWS IoT Core. You also receive this event when the connection resumes.

```
{
  "connectionStatusEvent": {
    "status": "CONNECTED"
```

```
  }
}
```

You receive the following example event when the core device loses its MQTT connection.

```
{
  "connectionStatusEvent": {
    "status": "DISCONNECTED"
  }
}
```

Examples

Use the following examples to learn how to use the AWS IoT Core MQTT IPC service in your components.

Example AWS IoT Core MQTT publisher (C++, IPC client V1)

The following example recipe allows the component to publish to all topics.

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.IoTCorePublisherCpp",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that publishes MQTT messages to IoT Core.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.mqttproxy": {
          "com.example.IoTCorePublisherCpp:mqttproxy:1": {
            "policyDescription": "Allows access to publish to all topics.",
            "operations": [
              "aws.greengrass#PublishToIoTCore"
            ],
            "resources": [
              "*"
            ]
          }
        }
```

```
      }
    }
  },
  "Manifests": [
    {
      "Lifecycle": {
        "Run": "{artifacts:path}/greengrassv2_iotcore_publisher"
      },
      "Artifacts": [
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.IoTCorePublisherCpp/1.0.0/greengrassv2_iotcore_publisher",
          "Permission": {
            "Execute": "OWNER"
          }
        }
      ]
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.IoTCorePublisherCpp
ComponentVersion: 1.0.0
ComponentDescription: A component that publishes MQTT messages to IoT Core.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    accessControl:
      aws.greengrass.ipc.mqttproxy:
        com.example.IoTCorePublisherCpp:mqttproxy:1:
          policyDescription: Allows access to publish to all topics.
          operations:
            - aws.greengrass#PublishToIoTCore
          resources:
            - "*"
Manifests:
  - Lifecycle:
      Run: "{artifacts:path}/greengrassv2_iotcore_publisher"
    Artifacts:
```

```
      - URI: s3://amzn-s3-demo-bucket/artifacts/
com.example.IoTCorePublisherCpp/1.0.0/greengrassv2_iotcore_publisher
        Permission:
          Execute: OWNER
```

The following example C++ application demonstrates how to use the AWS IoT Core MQTT IPC service to publish messages to AWS IoT Core.

```
#include <iostream>

#include <aws/crt/Api.h>
#include <aws/greengrass/GreengrassCoreIpcClient.h>

using namespace Aws::Crt;
using namespace Aws::Greengrass;

class IpcClientLifecycleHandler : public ConnectionLifecycleHandler {
    void OnConnectCallback() override {
        std::cout << "OnConnectCallback" << std::endl;
    }

    void OnDisconnectCallback(RpcError error) override {
        std::cout << "OnDisconnectCallback: " << error.StatusToString() << std::endl;
        exit(-1);
    }

    bool OnErrorCallback(RpcError error) override {
        std::cout << "OnErrorCallback: " << error.StatusToString() << std::endl;
        return true;
    }
};

int main() {
    String message("Hello from the Greengrass IPC MQTT publisher (C++).");
    String topic("test/topic/cpp");
    QOS qos = QOS_AT_LEAST_ONCE;
    int timeout = 10;

    ApiHandle apiHandle(g_allocator);
    Io::EventLoopGroup eventLoopGroup(1);
    Io::DefaultHostResolver socketResolver(eventLoopGroup, 64, 30);
    Io::ClientBootstrap bootstrap(eventLoopGroup, socketResolver);
```

```
    IpcClientLifecycleHandler ipcLifecycleHandler;
    GreengrassCoreIpcClient ipcClient(bootstrap);
    auto connectionStatus = ipcClient.Connect(ipcLifecycleHandler).get();
    if (!connectionStatus) {
        std::cerr << "Failed to establish IPC connection: " <<
connectionStatus.StatusToString() << std::endl;
        exit(-1);
    }

    while (true) {
        PublishToIoTCoreRequest request;
        Vector<uint8_t> messageData({message.begin(), message.end()});
        request.SetTopicName(topic);
        request.SetPayload(messageData);
        request.SetQos(qos);

        auto operation = ipcClient.NewPublishToIoTCore();
        auto activate = operation->Activate(request, nullptr);
        activate.wait();

        auto responseFuture = operation->GetResult();
        if (responseFuture.wait_for(std::chrono::seconds(timeout)) ==
std::future_status::timeout) {
            std::cerr << "Operation timed out while waiting for response from
Greengrass Core." << std::endl;
            exit(-1);
        }

        auto response = responseFuture.get();
        if (response) {
            std::cout << "Successfully published to topic: " << topic << std::endl;
        } else {
            // An error occurred.
            std::cout << "Failed to publish to topic: " << topic << std::endl;
            auto errorType = response.GetResultType();
            if (errorType == OPERATION_ERROR) {
                auto *error = response.GetOperationError();
                std::cout << "Operation error: " << error->GetMessage().value() <<
std::endl;
            } else {
                std::cout << "RPC error: " << response.GetRpcError() << std::endl;
            }
            exit(-1);
        }
```

```
        std::this_thread::sleep_for(std::chrono::seconds(5));
    }

    return 0;
}
```

Example AWS IoT Core MQTT subscriber (C++, IPC client V1)

The following example recipe allows the component to subscribe to all topics.

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.IoTCoreSubscriberCpp",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that subscribes to MQTT messages from IoT
 Core.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.mqttproxy": {
          "com.example.IoTCoreSubscriberCpp:mqttproxy:1": {
            "policyDescription": "Allows access to subscribe to all topics.",
            "operations": [
              "aws.greengrass#SubscribeToIoTCore"
            ],
            "resources": [
              "*"
            ]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Lifecycle": {
        "Run": "{artifacts:path}/greengrassv2_iotcore_subscriber"
      },
      "Artifacts": [
        {
```

```
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.IoTCoreSubscriberCpp/1.0.0/greengrassv2_iotcore_subscriber",
          "Permission": {
            "Execute": "OWNER"
          }
        }
      ]
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.IoTCoreSubscriberCpp
ComponentVersion: 1.0.0
ComponentDescription: A component that subscribes to MQTT messages from IoT Core.
ComponentPublisher: Amazon
ComponentConfiguration:
  DefaultConfiguration:
    accessControl:
      aws.greengrass.ipc.mqttproxy:
        com.example.IoTCoreSubscriberCpp:mqttproxy:1:
          policyDescription: Allows access to subscribe to all topics.
          operations:
            - aws.greengrass#SubscribeToIoTCore
          resources:
            - "*"
Manifests:
  - Lifecycle:
      Run: "{artifacts:path}/greengrassv2_iotcore_subscriber"
    Artifacts:
      - URI: s3://amzn-s3-demo-bucket/artifacts/
com.example.IoTCoreSubscriberCpp/1.0.0/greengrassv2_iotcore_subscriber
        Permission:
          Execute: OWNER
```

The following example C++ application demonstrates how to use the AWS IoT Core MQTT IPC service to subscribe to messages from AWS IoT Core.

```
#include <iostream>
```

```
#include <aws/crt/Api.h>
#include <aws/greengrass/GreengrassCoreIpcClient.h>

using namespace Aws::Crt;
using namespace Aws::Greengrass;

class IoTCoreResponseHandler : public SubscribeToIoTCoreStreamHandler {

    public:
        virtual ~IoTCoreResponseHandler() {}

    private:

        void OnStreamEvent(IoTCoreMessage *response) override {
            auto message = response->GetMessage();
            if (message.has_value() && message.value().GetPayload().has_value()) {
                auto messageBytes = message.value().GetPayload().value();
                std::string messageString(messageBytes.begin(), messageBytes.end());
                std::string messageTopic =
 message.value().GetTopicName().value().c_str();
                std::cout << "Received new message on topic: " << messageTopic <<
 std::endl;
                std::cout << "Message: " << messageString << std::endl;
            }
        }

        bool OnStreamError(OperationError *error) override {
            std::cout << "Received an operation error: ";
            if (error->GetMessage().has_value()) {
                std::cout << error->GetMessage().value();
            }
            std::cout << std::endl;
            return false; // Return true to close stream, false to keep stream open.
        }

        void OnStreamClosed() override {
            std::cout << "Subscribe to IoT Core stream closed." << std::endl;
        }
};

class IpcClientLifecycleHandler : public ConnectionLifecycleHandler {
    void OnConnectCallback() override {
        std::cout << "OnConnectCallback" << std::endl;
```

```
    }

    void OnDisconnectCallback(RpcError error) override {
        std::cout << "OnDisconnectCallback: " << error.StatusToString() << std::endl;
        exit(-1);
    }

    bool OnErrorCallback(RpcError error) override {
        std::cout << "OnErrorCallback: " << error.StatusToString() << std::endl;
        return true;
    }
};

int main() {
    String topic("test/topic/cpp");
    QOS qos = QOS_AT_LEAST_ONCE;
    int timeout = 10;

    ApiHandle apiHandle(g_allocator);
    Io::EventLoopGroup eventLoopGroup(1);
    Io::DefaultHostResolver socketResolver(eventLoopGroup, 64, 30);
    Io::ClientBootstrap bootstrap(eventLoopGroup, socketResolver);
    IpcClientLifecycleHandler ipcLifecycleHandler;
    GreengrassCoreIpcClient ipcClient(bootstrap);
    auto connectionStatus = ipcClient.Connect(ipcLifecycleHandler).get();
    if (!connectionStatus) {
        std::cerr << "Failed to establish IPC connection: " <<
 connectionStatus.StatusToString() << std::endl;
        exit(-1);
    }

    SubscribeToIoTCoreRequest request;
    request.SetTopicName(topic);
    request.SetQos(qos);
    auto streamHandler = MakeShared<IoTCoreResponseHandler>(DefaultAllocator());
    auto operation = ipcClient.NewSubscribeToIoTCore(streamHandler);
    auto activate = operation->Activate(request, nullptr);
    activate.wait();

    auto responseFuture = operation->GetResult();
    if (responseFuture.wait_for(std::chrono::seconds(timeout)) ==
 std::future_status::timeout) {
        std::cerr << "Operation timed out while waiting for response from Greengrass
 Core." << std::endl;
```

```
        exit(-1);
    }

    auto response = responseFuture.get();
    if (response) {
        std::cout << "Successfully subscribed to topic: " << topic << std::endl;
    } else {
        // An error occurred.
        std::cout << "Failed to subscribe to topic: " << topic << std::endl;
        auto errorType = response.GetResultType();
        if (errorType == OPERATION_ERROR) {
            auto *error = response.GetOperationError();
            std::cout << "Operation error: " << error->GetMessage().value() <<
 std::endl;
        } else {
            std::cout << "RPC error: " << response.GetRpcError() << std::endl;
        }
        exit(-1);
    }

    // Keep the main thread alive, or the process will exit.
    while (true) {
        std::this_thread::sleep_for(std::chrono::seconds(10));
    }

    operation->Close();
    return 0;
}
```

Example AWS IoT Core MQTT publisher (Rust)

The following example recipe allows the component to publish to all topics.

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.IoTCorePublisherRust",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that publishes MQTT messages to IoT Core.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.mqttproxy": {
          "com.example.IoTCorePublisherRust:mqttproxy:1": {
```

```
            "policyDescription": "Allows access to publish to all topics.",
            "operations": ["aws.greengrass#PublishToIoTCore"],
            "resources": ["*"]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux",
        "runtime": "*"
      },
      "Lifecycle": {
        "run": "{artifacts:path}/publish_to_iot_core"
      },
      "Artifacts": [
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.IoTCorePublisherRust/1.0.0/publish_to_iot_core",
          "Permission": {
            "Execute": "OWNER"
          }
        }
      ]
    }
  ]
}
```

The following example Rust application demonstrates how to use the AWS IoT Core MQTT IPC service to publish messages to AWS IoT Core.

```
use gg_sdk::{Qos, Sdk};

fn main() {
    let sdk = Sdk::init();
    sdk.connect().expect("Failed to establish IPC connection");

    let message = b"Hello, World";
    let topic = "my/topic";
    let qos = Qos::AtLeastOnce;
```

```
    sdk.publish_to_iot_core(topic, message, qos)
        .expect("Failed to publish to topic");

    println!("Successfully published to topic: {topic}");
}
```

Example AWS IoT Core MQTT subscriber (Rust)

The following example recipe allows the component to subscribe to all topics.

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.IoTCoreSubscriberRust",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that subscribes to MQTT messages from IoT
 Core.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.mqttproxy": {
          "com.example.IoTCoreSubscriberRust:mqttproxy:1": {
            "policyDescription": "Allows access to subscribe to all topics.",
            "operations": ["aws.greengrass#SubscribeToIoTCore"],
            "resources": ["*"]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux",
        "runtime": "*"
      },
      "Lifecycle": {
        "run": "{artifacts:path}/subscribe_to_iot_core"
      },
      "Artifacts": [
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.IoTCoreSubscriberRust/1.0.0/subscribe_to_iot_core",
          "Permission": {
```

```
            "Execute": "OWNER"
          }
        }
      ]
    }
  ]
}
```

The following example Rust application demonstrates how to use the AWS IoT Core MQTT IPC service to subscribe to messages from AWS IoT Core.

```
use gg_sdk::{Qos, Sdk};
use std::{thread, time::Duration};

fn main() {
    let sdk = Sdk::init();
    sdk.connect().expect("Failed to establish IPC connection");

    let topic = "my/topic";
    let qos = Qos::AtLeastOnce;

    let callback = |topic: &str, payload: &[u8]| {
        let message = String::from_utf8_lossy(payload);
        println!("Received new message on topic {topic}: {message}");
    };

    let _sub = sdk
        .subscribe_to_iot_core(topic, qos, &callback)
        .expect("Failed to subscribe to topic");

    println!("Successfully subscribed to topic: {topic}");

    // Keep the main thread alive, or the process will exit.
    loop {
        thread::sleep(Duration::from_secs(10));
    }
}
```

Example AWS IoT Core MQTT publisher (C)

The following example recipe allows the component to publish to all topics.

```
{
```

```
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.IoTCorePublisherC",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that publishes MQTT messages to IoT Core.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.mqttproxy": {
          "com.example.IoTCorePublisherC:mqttproxy:1": {
            "policyDescription": "Allows access to publish to all topics.",
            "operations": ["aws.greengrass#PublishToIoTCore"],
            "resources": ["*"]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux",
        "runtime": "*"
      },
      "Lifecycle": {
        "run": "{artifacts:path}/sample_publish_to_iot_core"
      },
      "Artifacts": [
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.IoTCorePublisherC/1.0.0/sample_publish_to_iot_core",
          "Permission": {
            "Execute": "OWNER"
          }
        }
      ]
    }
  ]
}
```

The following example C application demonstrates how to use the AWS IoT Core MQTT IPC service to publish messages to AWS IoT Core.

```
#include <gg/error.h>
#include <gg/ipc/client.h>
#include <gg/sdk.h>
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    gg_sdk_init();

    GgError err = ggipc_connect();
    if (err != GG_ERR_OK) {
        fprintf(stderr, "Failed to establish IPC connection.\n");
        exit(-1);
    }

    GgBuffer message = GG_STR("Hello, World");
    GgBuffer topic = GG_STR("my/topic");
    uint8_t qos = 1;

    err = ggipc_publish_to_iot_core(topic, message, qos);
    if (err != GG_ERR_OK) {
        fprintf(
            stderr,
            "Failed to publish to topic: %.*s\n",
            (int) topic.len,
            topic.data
        );
        exit(-1);
    }

    printf(
        "Successfully published to topic: %.*s\n", (int) topic.len, topic.data
    );
}
```

Example AWS IoT Core MQTT subscriber (C)

The following example recipe allows the component to subscribe to all topics.

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.IoTCoreSubscriberC",
  "ComponentVersion": "1.0.0",
```

```
  "ComponentDescription": "A component that subscribes to MQTT messages from IoT
 Core.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.mqttproxy": {
          "com.example.IoTCoreSubscriberC:mqttproxy:1": {
            "policyDescription": "Allows access to subscribe to all topics.",
            "operations": ["aws.greengrass#SubscribeToIoTCore"],
            "resources": ["*"]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux",
        "runtime": "*"
      },
      "Lifecycle": {
        "run": "{artifacts:path}/sample_subscribe_to_iot_core"
      },
      "Artifacts": [
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.IoTCoreSubscriberC/1.0.0/sample_subscribe_to_iot_core",
          "Permission": {
            "Execute": "OWNER"
          }
        }
      ]
    }
  ]
}
```

The following example C application demonstrates how to use the AWS IoT Core MQTT IPC service to subscribe to messages from AWS IoT Core.

```
#include <gg/error.h>
#include <gg/ipc/client.h>
```

```
#include <gg/sdk.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

static void on_subscription_response(
    void *ctx, GgBuffer topic, GgBuffer payload, GgIpcSubscriptionHandle handle
) {
    (void) ctx;
    (void) handle;

    printf(
        "Received new message on topic %.*s: %.*s\n",
        (int) topic.len,
        topic.data,
        (int) payload.len,
        payload.data
    );
}

int main(void) {
    gg_sdk_init();

    GgError err = ggipc_connect();
    if (err != GG_ERR_OK) {
        fprintf(stderr, "Failed to establish IPC connection.\n");
        exit(-1);
    }

    GgBuffer topic = GG_STR("my/topic");
    uint8_t qos = 1;

    GgIpcSubscriptionHandle handle;
    err = ggipc_subscribe_to_iot_core(
        topic, qos, on_subscription_response, NULL, &handle
    );
    if (err != GG_ERR_OK) {
        fprintf(
            stderr,
            "Failed to subscribe to topic: %.*s\n",
            (int) topic.len,
            topic.data
        );
        exit(-1);
```

```
    }

    printf(
        "Successfully subscribed to topic: %.*s\n", (int) topic.len, topic.data
    );

    // Keep the main thread alive, or the process will exit.
    while (1) {
        sleep(10);
    }

    // To stop subscribing, close the subscription handle.
    ggipc_close_subscription(handle);
}
```

Example AWS IoT Core MQTT publisher (C++, Component SDK)

The following example recipe allows the component to publish to all topics.

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.IoTCorePublisherCpp",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that publishes MQTT messages to IoT Core.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.mqttproxy": {
          "com.example.IoTCorePublisherCpp:mqttproxy:1": {
            "policyDescription": "Allows access to publish to all topics.",
            "operations": ["aws.greengrass#PublishToIoTCore"],
            "resources": ["*"]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux",
        "runtime": "*"
      },
```

```
      "Lifecycle": {
        "run": "{artifacts:path}/sample_cpp_publish_to_iot_core"
      },
      "Artifacts": [
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.IoTCorePublisherCpp/1.0.0/sample_cpp_publish_to_iot_core",
          "Permission": {
            "Execute": "OWNER"
          }
        }
      ]
    }
  ]
}
```

The following example C++ application demonstrates how to use the AWS IoT Core MQTT IPC service to publish messages to AWS IoT Core.

```
#include <gg/ipc/client.hpp>
#include <iostream>

int main() {
    auto &client = gg::ipc::Client::get();

    auto error = client.connect();
    if (error) {
        std::cerr << "Failed to establish IPC connection.\n";
        exit(-1);
    }

    std::string_view message = "Hello, World";
    std::string_view topic = "my/topic";
    uint8_t qos = 1;

    error = client.publish_to_iot_core(topic, message, qos);
    if (error) {
        std::cerr << "Failed to publish to topic: " << topic << "\n";
        exit(-1);
    }

    std::cout << "Successfully published to topic: " << topic << "\n";
}
```

Example AWS IoT Core MQTT subscriber (C++, Component SDK)

The following example recipe allows the component to subscribe to all topics.

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.IoTCoreSubscriberCpp",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "A component that subscribes to MQTT messages from IoT
 Core.",
  "ComponentPublisher": "Amazon",
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "accessControl": {
        "aws.greengrass.ipc.mqttproxy": {
          "com.example.IoTCoreSubscriberCpp:mqttproxy:1": {
            "policyDescription": "Allows access to subscribe to all topics.",
            "operations": ["aws.greengrass#SubscribeToIoTCore"],
            "resources": ["*"]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux",
        "runtime": "*"
      },
      "Lifecycle": {
        "run": "{artifacts:path}/sample_cpp_subscribe_to_iot_core"
      },
      "Artifacts": [
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.IoTCoreSubscriberCpp/1.0.0/sample_cpp_subscribe_to_iot_core",
          "Permission": {
            "Execute": "OWNER"
          }
        }
      ]
    }
  ]
```

```
}
```

The following example C++ application demonstrates how to use the AWS IoT Core MQTT IPC service to subscribe to messages from AWS IoT Core.

```
#include <gg/ipc/client.hpp>
#include <unistd.h>
#include <iostream>

class ResponseHandler : public gg::ipc::IotTopicCallback {
    void operator()(
        std::string_view topic,
        gg::Buffer payload,
        gg::ipc::Subscription &handle
    ) override {
        (void) handle;
        std::cout << "Received new message on topic " << topic << ": "
                  << payload << "\n";
    }
};

int main() {
    auto &client = gg::ipc::Client::get();

    auto error = client.connect();
    if (error) {
        std::cerr << "Failed to establish IPC connection.\n";
        exit(-1);
    }

    std::string_view topic = "my/topic";
    uint8_t qos = 1;

    static ResponseHandler handler;
    error = client.subscribe_to_iot_core(topic, qos, handler);
    if (error) {
        std::cerr << "Failed to subscribe to topic: " << topic << "\n";
        exit(-1);
    }

    std::cout << "Successfully subscribed to topic: " << topic << "\n";

    // Keep the main thread alive, or the process will exit.
```

```
    while (1) {
        sleep(10);
    }
}
```

Interact with component lifecycle

Use the component lifecycle IPC service to:

- Update the component state on the core device.
- Subscribe to component state updates.
- Prevent the nucleus from stopping the component to apply an update during a deployment.
- Pause and resume component processes.

Topics

- Minimum SDK versions
- Authorization
- UpdateState
- SubscribeToComponentUpdates
- DeferComponentUpdate
- PauseComponent
- ResumeComponent

Minimum SDK versions

The following table lists the minimum versions of the SDKs that you can use to interact with component lifecycle.

SDK	Minimum version	
AWS IoT Greengrass Component SDK (C, C++, Rust)	v1.0.0	

SDK	Minimum version	
AWS IoT Device SDK for Java v2	v1.2.10	
AWS IoT Device SDK for Python v2	v1.5.3	
AWS IoT Device SDK for C++ v2	v1.17.0	
AWS IoT Device SDK for JavaScript v2	v1.12.0	

Authorization

To pause or resume other components from a custom component, you must define authorization policies that allow your component to manage other components. For information about defining authorization policies, see Authorize components to perform IPC operations.

Authorization policies for component lifecycle management have the following properties.

IPC service identifier: `aws.greengrass.ipc.lifecycle`

Operation	Description	Resources
`aws.greengrass#PauseComponent`	Allows a component to pause the components that you specify.	A component name, or * to allow access to all components.
`aws.greengrass#ResumeComponent`	Allows a component to resume the components that you specify.	A component name, or * to allow access to all components.
*	Allows a component to pause and resume the components that you specify.	A component name, or * to allow access to all components.

Authorization policy examples

You can reference the following authorization policy example to help you configure authorization policies for your components.

Example authorization policy

The following example authorization policy allows a component to pause and resume all components.

```
{
  "accessControl": {
    "aws.greengrass.ipc.lifecycle": {
      "com.example.MyLocalLifecycleComponent:lifecycle:1": {
        "policyDescription": "Allows access to pause/resume all components.",
        "operations": [
          "aws.greengrass#PauseComponent",
          "aws.greengrass#ResumeComponent"
        ],
        "resources": [
          "*"
        ]
      }
    }
  }
}
```

UpdateState

Update the state of the component on the core device.

Request

This operation's request has the following parameters:

`state`

The state to set. This enum, `LifecycleState`, has the following values:

- `RUNNING`
- `ERRORED`

Response

This operation doesn't provide any information in its response.

Examples

The following examples demonstrate how to call this operation in custom component code.

Rust

Example: Update state

```
use gg_sdk::{ComponentState, Sdk};

fn main() {
    let sdk = Sdk::init();
    sdk.connect().expect("Failed to establish IPC connection");

    // Update component state to RUNNING
    sdk.update_state(ComponentState::Running)
        .expect("Failed to update component state");

    println!("Successfully updated component state to RUNNING.");
}
```

C

Example: Update state

```
#include <gg/error.h>
#include <gg/ipc/client.h>
#include <gg/sdk.h>
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    gg_sdk_init();

    GgError err = ggipc_connect();
    if (err != GG_ERR_OK) {
        fprintf(stderr, "Failed to establish IPC connection.\n");
        exit(-1);
```

```
    }

    // Update component state to RUNNING
    err = ggipc_update_state(GG_COMPONENT_STATE_RUNNING);
    if (err != GG_ERR_OK) {
        fprintf(stderr, "Failed to update component state.\n");
        exit(-1);
    }

    printf("Successfully updated component state to RUNNING.\n");
}
```

C++ (Component SDK)

Example: Update state

```
#include <gg/ipc/client.hpp>
#include <iostream>

int main() {
    auto &client = gg::ipc::Client::get();

    auto error = client.connect();
    if (error) {
        std::cerr << "Failed to establish IPC connection.\n";
        exit(-1);
    }

    // Update component state to RUNNING
    error = client.update_component_state(GG_COMPONENT_STATE_RUNNING);
    if (error) {
        std::cerr << "Failed to update component state.\n";
        exit(-1);
    }

    std::cout << "Successfully updated component state to RUNNING.\n";
}
```

SubscribeToComponentUpdates

Subscribe to receive notifications before the AWS IoT Greengrass Core software updates a component. The notification specifies whether or not the nucleus will restart as part of the update.

The nucleus sends update notifications only if the deployment's component update policy specifies to notify components. The default behavior is to notify components. For more information, see Create deployments and the DeploymentComponentUpdatePolicy object that you can provide when you call the CreateDeployment operation.

> **Important**
>
> Local deployments don't notify components before updates.

This operation is a subscription operation where you subscribe to a stream of event messages. To use this operation, define a stream response handler with functions that handle event messages, errors, and stream closure. For more information, see Subscribe to IPC event streams.

Event message type: `ComponentUpdatePolicyEvents`

> **Tip**
>
> You can follow a tutorial to learn how to develop a component that conditionally defers component updates. For more information, see Tutorial: Develop a Greengrass component that defers component updates.

Request

This operation's request doesn't have any parameters.

Response

This operation's response has the following information:

`messages`

The stream of notification messages. This object, `ComponentUpdatePolicyEvents`, contains the following information:

`preUpdateEvent` (Python: `pre_update_event`)

(Optional) An event that indicates that the nucleus wants to update a component. You can respond with the DeferComponentUpdate operation to acknowledge or defer the

update until your component is ready to restart. This object, `PreComponentUpdateEvent`, contains the following information:

`deploymentId` (Python: `deployment_id`)

The ID of the AWS IoT Greengrass deployment that updates the component.

`isGgcRestarting` (Python: `is_ggc_restarting`)

Whether or not the nucleus needs to restart to apply the update.

`postUpdateEvent` (Python: `post_update_event`)

(Optional) An event that indicates that the nucleus updated a component. This object, `PostComponentUpdateEvent`, contains the following information:

`deploymentId` (Python: `deployment_id`)

The ID of the AWS IoT Greengrass deployment that updated the component.

> **Note**
>
> This feature requires v2.7.0 or later of the Greengrass nucleus component.

DeferComponentUpdate

Acknowledge or defer a component update that you discover with SubscribeToComponentUpdates. You specify the amount of time to wait before the nucleus checks again if your component is ready to let the component update proceed. You can also use this operation to tell the nucleus that your component is ready for the update.

If a component doesn't respond to the component update notification, the nucleus waits the amount of time that you specify in the deployment's component update policy. After that timeout, the nucleus proceeds with the deployment. The default component update timeout is 60 seconds. For more information, see Create deployments and the DeploymentComponentUpdatePolicy object that you can provide when you call the CreateDeployment operation.

Tip

You can follow a tutorial to learn how to develop a component that conditionally defers component updates. For more information, see Tutorial: Develop a Greengrass component that defers component updates.

Request

This operation's request has the following parameters:

`deploymentId` (Python: `deployment_id`)

The ID of the AWS IoT Greengrass deployment to defer.

`message`

(Optional) The name of the component for which to defer updates.

Defaults to the name of the component that makes the request.

`recheckAfterMs` (Python: `recheck_after_ms`)

The amount of time in milliseconds for which to defer the update. The nucleus waits for this amount of time and then sends another `PreComponentUpdateEvent` that you can discover with SubscribeToComponentUpdates.

Specify `0` to acknowledge the update. This tells the nucleus that your component is ready for the update.

Defaults to zero milliseconds, which means to acknowledge the update.

Response

This operation doesn't provide any information in its response.

PauseComponent

This feature is available for v2.4.0 and later of the Greengrass nucleus component. AWS IoT Greengrass doesn't currently support this feature on Windows core devices.

Pauses a component's processes on the core device. To resume a component, use the ResumeComponent operation.

You can pause only generic components. If you try to pause any other type of component, this operation throws an `InvalidRequestError`.

> **Note**
>
> This operation can't pause containerized processes, such as Docker containers. To pause and resume a Docker container, you can use the docker pause and docker unpause commands.

This operation doesn't pause component dependencies or components that depend on the component that you pause. Consider this behavior when you pause a component that is a dependency of another component, because the dependent component might encounter issues when its dependency is paused.

When you restart or shut down a paused component, such as through a deployment, the Greengrass nucleus resumes the component and runs its shutdown lifecycle. For more information about restarting a component, see RestartComponent.

> **Important**
>
> To use this operation, you must define an authorization policy that grants permission to use this operation. For more information, see Authorization.

Minimum SDK versions

The following table lists the minimum versions of the SDKs that you can use to pause and resume components.

SDK	Minimum version
AWS IoT Device SDK for Java v2	v1.4.3
AWS IoT Device SDK for Python v2	v1.6.2
AWS IoT Device SDK for C++ v2	v1.13.1

SDK	Minimum version	
AWS IoT Device SDK for JavaScript v2	v1.12.0	

Request

This operation's request has the following parameters:

`componentName` (Python: `component_name`)

The name of the component to pause, which must be a generic component. For more information, see Component types.

Response

This operation doesn't provide any information in its response.

ResumeComponent

This feature is available for v2.4.0 and later of the Greengrass nucleus component. AWS IoT Greengrass doesn't currently support this feature on Windows core devices.

Resumes a component's processes on the core device. To pause a component, use the PauseComponent operation.

You can resume only paused components. If you try to resume a component that isn't paused, this operation throws an `InvalidRequestError`.

Important

To use this operation, you must define an authorization policy that grants permission to do so. For more information, see Authorization.

Minimum SDK versions

The following table lists the minimum versions of the SDKs that you can use to pause and resume components.

SDK	Minimum version	
AWS IoT Device SDK for Java v2	v1.4.3	
AWS IoT Device SDK for Python v2	v1.6.2	
AWS IoT Device SDK for C++ v2	v1.13.1	
AWS IoT Device SDK for JavaScript v2	v1.12.0	

Request

This operation's request has the following parameters:

`componentName` (Python: `component_name`)

The name of the component to resume.

Response

This operation doesn't provide any information in its response.

Interact with component configuration

The component configuration IPC service lets you do the following:

- Get and set component configuration parameters.
- Subscribe to component configuration updates.
- Validate component configuration updates before the nucleus applies them.

Topics

- Minimum SDK versions

- GetConfiguration
- UpdateConfiguration
- SubscribeToConfigurationUpdate
- SubscribeToValidateConfigurationUpdates
- SendConfigurationValidityReport

Minimum SDK versions

The following table lists the minimum versions of the SDKs that you can use to interact with component configuration.

SDK	Minimum version	
AWS IoT Greengrass Component SDK (C, C++, Rust)	v1.0.0	
AWS IoT Device SDK for Java v2	v1.2.10	
AWS IoT Device SDK for Python v2	v1.5.3	
AWS IoT Device SDK for C++ v2	v1.17.0	
AWS IoT Device SDK for JavaScript v2	v1.12.0	

GetConfiguration

Gets a configuration value for a component on the core device. You specify the key path for which to get a configuration value.

Request

This operation's request has the following parameters:

`componentName` (Python: `component_name`)

(Optional) The name of the component.

Defaults to the name of the component that makes the request.

`keyPath` (Python: `key_path`)

The key path to the configuration value. Specify a list where each entry is the key for a single level in the configuration object. For example, specify `["mqtt", "port"]` to get the value of `port` in the following configuration.

```
{
  "mqtt": {
    "port": 443
  }
}
```

To get the component's complete configuration, specify an empty list.

Response

This operation's response has the following information:

`componentName` (Python: `component_name`)

The name of the component.

`value`

The requested configuration as an object.

Examples

The following examples demonstrate how to call this operation in custom component code.

Rust

Example: Get configuration

```
use core::mem::MaybeUninit;
use gg_sdk::{Sdk, UnpackedObject};
```

```
fn main() {
    let sdk = Sdk::init();
    sdk.connect().expect("Failed to establish IPC connection");

    // Get a configuration value at key path ["mqtt", "port"]
    let mut buf = [MaybeUninit::uninit(); 1024];

    let value = sdk
        .get_config(&["mqtt", "port"], None, &mut buf)
        .expect("Failed to get configuration");

    if let UnpackedObject::I64(port) = value.unpack() {
        println!("Configuration value: {port}");
    }
}
```

C

Example: Get configuration

```
#include <gg/error.h>
#include <gg/ipc/client.h>
#include <gg/object.h>
#include <gg/sdk.h>
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    gg_sdk_init();

    GgError err = ggipc_connect();
    if (err != GG_ERR_OK) {
        fprintf(stderr, "Failed to establish IPC connection.\n");
        exit(-1);
    }

    // Get a configuration value at key path ["mqtt", "port"]
    uint8_t response_mem[1024];
    GgObject value;

    err = ggipc_get_config(
        GG_BUF_LIST(GG_STR("mqtt"), GG_STR("port")),
```

```
        NULL, // component_name (NULL = current component)
        GG_BUF(response_mem),
        &value
    );
    if (err != GG_ERR_OK) {
        fprintf(stderr, "Failed to get configuration.\n");
        exit(-1);
    }

    if (gg_obj_type(value) == GG_TYPE_I64) {
        printf("Configuration value: %" PRId64 "\n", gg_obj_into_i64(value));
    } else if (gg_obj_type(value) == GG_TYPE_BUF) {
        GgBuffer buf = gg_obj_into_buf(value);
        printf("Configuration value: %.*s\n", (int) buf.len, buf.data);
    } else {
        printf("Configuration value is of unexpected type.\n");
    }
}
```

C++ (Component SDK)

Example: Get configuration

```
#include <gg/ipc/client.hpp>
#include <iostream>

int main() {
    auto &client = gg::ipc::Client::get();

    auto error = client.connect();
    if (error) {
        std::cerr << "Failed to establish IPC connection.\n";
        exit(-1);
    }

    // Get a configuration value at key path ["mqtt", "port"]
    std::array key_path = { gg::Buffer { "mqtt" }, gg::Buffer { "port" } };
    int64_t value = 0;

    error = client.get_config(key_path, std::nullopt, value);
    if (error) {
        std::cerr << "Failed to get configuration.\n";
        exit(-1);
    }
```

```
    std::cout << "Configuration value: " << value << "\n";
}
```

UpdateConfiguration

Updates a configuration value for this component on the core device.

Request

This operation's request has the following parameters:

`keyPath` (Python: `key_path`)

(Optional) The key path to the container node (the object) to update. Specify a list where each entry is the key for a single level in the configuration object. For example, specify the key path `["mqtt"]` and the merge value `{ "port": 443 }` to set the value of `port` in the following configuration.

```
{
  "mqtt": {
    "port": 443
  }
}
```

The key path must specify a container node (an object) in the configuration. If the node doesn't exist in the component's configuration, this operation creates it and sets its value to the object in `valueToMerge`.

Defaults to the root of the configuration object.

`timestamp`

The current Unix epoch time in milliseconds. This operation uses this timestamp to resolve concurrent updates to the key. If the key in the component configuration has a greater timestamp than the timestamp in the request, then the request fails.

`valueToMerge` (Python: `value_to_merge`)

The configuration object to merge at the location that you specify in `keyPath`. For more information, see Update component configurations.

Response

This operation doesn't provide any information in its response.

Examples

The following examples demonstrate how to call this operation in custom component code.

Rust

Example: Update configuration

```
use gg_sdk::Sdk;

fn main() {
    let sdk = Sdk::init();
    sdk.connect().expect("Failed to establish IPC connection");

    // Update configuration value at key path ["mqtt", "port"] to 443
    sdk.update_config(&["mqtt", "port"], None, 443)
        .expect("Failed to update configuration");

    println!("Successfully updated configuration.");
}
```

C

Example: Update configuration

```
#include <gg/error.h>
#include <gg/ipc/client.h>
#include <gg/object.h>
#include <gg/sdk.h>
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    gg_sdk_init();

    GgError err = ggipc_connect();
    if (err != GG_ERR_OK) {
        fprintf(stderr, "Failed to establish IPC connection.\n");
```

```
        exit(-1);
    }

    // Update configuration value at key path ["mqtt", "port"] to 443
    err = ggipc_update_config(
        GG_BUF_LIST(GG_STR("mqtt"), GG_STR("port")),
        NULL, // timestamp (NULL = current time)
        gg_obj_i64(443)
    );
    if (err != GG_ERR_OK) {
        fprintf(stderr, "Failed to update configuration.\n");
        exit(-1);
    }

    printf("Successfully updated configuration.\n");
}
```

C++ (Component SDK)

Example: Update configuration

```
#include <gg/ipc/client.hpp>
#include <iostream>

int main() {
    auto &client = gg::ipc::Client::get();

    auto error = client.connect();
    if (error) {
        std::cerr << "Failed to establish IPC connection.\n";
        exit(-1);
    }

    // Update configuration value at key path ["mqtt", "port"] to 443
    std::array key_path = { gg::Buffer { "mqtt" }, gg::Buffer { "port" } };

    error = client.update_config(key_path, 443);
    if (error) {
        std::cerr << "Failed to update configuration.\n";
        exit(-1);
    }

    std::cout << "Successfully updated configuration.\n";
```

```
}
```

SubscribeToConfigurationUpdate

Subscribe to receive notifications when a component's configuration updates. When you subscribe to a key, you receive a notification when any child of that key updates.

This operation is a subscription operation where you subscribe to a stream of event messages. To use this operation, define a stream response handler with functions that handle event messages, errors, and stream closure. For more information, see Subscribe to IPC event streams.

Event message type: `ConfigurationUpdateEvents`

Request

This operation's request has the following parameters:

`componentName` (Python: `component_name`)

(Optional) The name of the component.

Defaults to the name of the component that makes the request.

`keyPath` (Python: `key_path`)

The key path to the configuration value for which to subscribe. Specify a list where each entry is the key for a single level in the configuration object. For example, specify `["mqtt", "port"]` to get the value of `port` in the following configuration.

```
{
  "mqtt": {
    "port": 443
  }
}
```

To subscribe to updates for all values in the component's configuration, specify an empty list.

Response

This operation's response has the following information:

`messages`

The stream of notification messages. This object, `ConfigurationUpdateEvents`, contains the following information:

`configurationUpdateEvent` (Python: `configuration_update_event`)

The configuration update event. This object, `ConfigurationUpdateEvent`, contains the following information:

`componentName` (Python: `component_name`)

The name of the component.

`keyPath` (Python: `key_path`)

The key path to the configuration value that updated.

Examples

The following examples demonstrate how to call this operation in custom component code.

Rust

Example: Subscribe to configuration updates

```
use gg_sdk::Sdk;
use std::{thread, time::Duration};

fn main() {
    let sdk = Sdk::init();
    sdk.connect().expect("Failed to establish IPC connection");

    // Subscribe to configuration updates for key path ["mqtt"]
    let callback = |component_name: &str, key_path: &[&str]| {
        println!(
            "Received configuration update for component: {component_name}"
        );
        println!("Key path: {key_path:?}");
    };

    let _sub = sdk
        .subscribe_to_configuration_update(None, &["mqtt"], &callback)
        .expect("Failed to subscribe to configuration updates");
```

```
    println!("Successfully subscribed to configuration updates.");

    // Keep the main thread alive, or the process will exit.
    loop {
        thread::sleep(Duration::from_secs(10));
    }
}
```

C

Example: Subscribe to configuration updates

```
#include <gg/error.h>
#include <gg/ipc/client.h>
#include <gg/object.h>
#include <gg/sdk.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

static void on_subscription_response(
    void *ctx,
    GgBuffer component_name,
    GgList key_path,
    GgIpcSubscriptionHandle handle
) {
    (void) ctx;
    (void) handle;

    printf(
        "Received configuration update for component: %.*s\n",
        (int) component_name.len,
        component_name.data
    );

    printf("Key path: [");
    for (size_t i = 0; i < key_path.len; i++) {
        if (i > 0) {
            printf(", ");
        }
        GgObject *obj = &key_path.items[i];
        if (gg_obj_type(*obj) == GG_TYPE_BUF) {
            GgBuffer key = gg_obj_into_buf(*obj);
            printf("\"%.*s\"", (int) key.len, key.data);
```

```
        }
    }
    printf("]\n");
}

int main(void) {
    gg_sdk_init();

    GgError err = ggipc_connect();
    if (err != GG_ERR_OK) {
        fprintf(stderr, "Failed to establish IPC connection.\n");
        exit(-1);
    }

    // Subscribe to configuration updates for key path ["mqtt"]
    GgIpcSubscriptionHandle handle;
    err = ggipc_subscribe_to_configuration_update(
        NULL, // component_name (NULL = current component)
        GG_BUF_LIST(GG_STR("mqtt")),
        on_subscription_response,
        NULL,
        &handle
    );
    if (err != GG_ERR_OK) {
        fprintf(stderr, "Failed to subscribe to configuration updates.\n");
        exit(-1);
    }

    printf("Successfully subscribed to configuration updates.\n");

    // Keep the main thread alive, or the process will exit.
    while (1) {
        sleep(10);
    }

    // To stop subscribing, close the stream.
    ggipc_close_subscription(handle);
}
```

C++ (Component SDK)

Example: Subscribe to configuration updates

```
#include <gg/ipc/client.hpp>
```

```
#include <unistd.h>
#include <iostream>

class ResponseHandler : public gg::ipc::ConfigurationUpdateCallback {
    void operator()(
        std::string_view component_name,
        gg::List key_path,
        gg::ipc::Subscription &handle
    ) override {
        (void) handle;
        std::cout << "Received configuration update for component: "
                  << component_name << "\n";
        std::cout << "Key path: [";
        for (size_t i = 0; i < key_path.size(); i++) {
            if (i > 0) {
                std::cout << ", ";
            }
            std::cout << "\"" << get<gg::Buffer>(key_path[i]) << "\"";
        }
        std::cout << "]\n";
    }
};

int main() {
    auto &client = gg::ipc::Client::get();

    auto error = client.connect();
    if (error) {
        std::cerr << "Failed to establish IPC connection.\n";
        exit(-1);
    }

    // Subscribe to configuration updates for key path ["mqtt"]
    std::array key_path = { gg::Buffer { "mqtt" } };

    static ResponseHandler handler;
    error = client.subscribe_to_configuration_update(
        key_path, std::nullopt, handler
    );
    if (error) {
        std::cerr << "Failed to subscribe to configuration updates.\n";
        exit(-1);
    }
```

```
    std::cout << "Successfully subscribed to configuration updates.\n";

    // Keep the main thread alive, or the process will exit.
    while (1) {
        sleep(10);
    }
}
```

SubscribeToValidateConfigurationUpdates

Subscribe to receive notifications before this component's configuration updates. This lets components validate updates to their own configuration. Use the SendConfigurationValidityReport operation to tell the nucleus whether or not the configuration is valid.

> **Important**
>
> Local deployments don't notify components of updates.

This operation is a subscription operation where you subscribe to a stream of event messages. To use this operation, define a stream response handler with functions that handle event messages, errors, and stream closure. For more information, see Subscribe to IPC event streams.

Event message type: `ValidateConfigurationUpdateEvents`

Request

This operation's request doesn't have any parameters.

Response

This operation's response has the following information:

`messages`

The stream of notification messages. This object, `ValidateConfigurationUpdateEvents`, contains the following information:

`validateConfigurationUpdateEvent` (Python: `validate_configuration_update_event`)

The configuration update event. This object, `ValidateConfigurationUpdateEvent`, contains the following information:

`deploymentId` (Python: `deployment_id`)

The ID of the AWS IoT Greengrass deployment that updates the component.

`configuration`

The object that contains the new configuration.

SendConfigurationValidityReport

Tell the nucleus whether or not a configuration update to this component is valid. The deployment fails if you tell the nucleus that the new configuration isn't valid. Use the SubscribeToValidateConfigurationUpdates operation to subscribe to validate configuration updates.

If a component doesn't respond to a validate configuration update notification, the nucleus waits the amount of time that you specify in the deployment's configuration validation policy. After that timeout, the nucleus proceeds with the deployment. The default component validation timeout is 20 seconds. For more information, see Create deployments and the DeploymentConfigurationValidationPolicy object that you can provide when you call the CreateDeployment operation.

Request

This operation's request has the following parameters:

`configurationValidityReport` (Python: `configuration_validity_report`)

The report that tells the nucleus whether or not the configuration update is valid. This object, `ConfigurationValidityReport`, contains the following information:

`status`

The validity status. This enum, `ConfigurationValidityStatus`, has the following values:

- `ACCEPTED` – The configuration is valid and the nucleus can apply it to this component.

- `REJECTED` – The configuration isn't valid and the deployment fails.

`deploymentId` (Python: `deployment_id`)

The ID of the AWS IoT Greengrass deployment that requested the configuration update.

`message`

(Optional) A message that reports why the configuration isn't valid.

Response

This operation doesn't provide any information in its response.

Retrieve secret values

Use the secret manager IPC service to retrieve secret values from secrets on the core device. You use the secret manager component to deploy encrypted secrets to core devices. Then, you can use an IPC operation to decrypt the secret and use its value in your custom components.

Topics

- Minimum SDK versions
- Authorization
- GetSecretValue
- Examples

Minimum SDK versions

The following table lists the minimum versions of the AWS IoT Device SDK that you must use to retrieve secret values from secrets on the core device.

SDK	Minimum version	
AWS IoT Device SDK for Java v2	v1.2.10	
AWS IoT Device SDK for Python v2	v1.5.3	

SDK	Minimum version	
AWS IoT Device SDK for C++ v2	v1.17.0	
AWS IoT Device SDK for JavaScript v2	v1.12.0	

Authorization

To use secret manager in a custom component, you must define authorization policies that allow your component to get the value of secrets that you store on the core device. For information about defining authorization policies, see Authorize components to perform IPC operations.

Authorization policies for secret manager have the following properties.

IPC service identifier: `aws.greengrass.SecretManager`

Operation	Description	Resources
`aws.greengrass#GetSecretValue`	Allows a component to get the value of secrets that are encrypted on the core device.	A Secrets Manager secret ARN, or `*` to allow access to all secrets.

Authorization policy examples

You can reference the following authorization policy example to help you configure authorization policies for your components.

Example authorization policy

The following example authorization policy allows a component to get the value of any secret on the core device.

> **Note**
>
> We recommend that in a production environment, you reduce the scope of the authorization policy, so that the component retrieves only the secrets that it uses. You can change the * wildcard to a list of secret ARNs when you deploy the component.

```
{
  "accessControl": {
    "aws.greengrass.SecretManager": {
      "com.example.MySecretComponent:secrets:1": {
        "policyDescription": "Allows access to a secret.",
        "operations": [
          "aws.greengrass#GetSecretValue"
        ],
        "resources": [
          "*"
        ]
      }
    }
  }
}
```

GetSecretValue

Gets the value of a secret that you store on the core device.

This operation is similar to the Secrets Manager operation that you can use to get the value of a secret in the AWS Cloud. For more information, see GetSecretValue in the *AWS Secrets Manager API Reference*.

Request

This operation's request has the following parameters:

`refresh` (Python: `refresh`)

(optional): Whether to sync the requested secret with its latest value from AWS Secrets Manager service.

When set to true, secret manager will request the AWS Secrets Manager service for the latest value of the specified secret label and returns that value as a response. Otherwise, the secret value that was stored locally will be returned.

This parameter will not work in conjunction with `versionId` parameter in the request. This parameter works when used in conjunction with Nucleus 2.13.0 and above.

`secretId` (Python: `secret_id`)

The name of the secret to get. You can specify either the Amazon Resource Name (ARN) or the friendly name of the secret.

`versionId` (Python: `version_id`)

(Optional) The ID of the version to get.

You can specify either `versionId` or `versionStage`.

If you don't specify `versionId` or `versionStage`, this operation defaults to the version with the `AWSCURRENT` label.

`versionStage` (Python: `version_stage`)

(Optional) The staging label of the version to get.

You can specify either `versionId` or `versionStage`.

If you don't specify `versionId` or `versionStage`, this operation defaults to the version with the `AWSCURRENT` label.

Response

This operation's response has the following information:

`secretId` (Python: `secret_id`)

The ID of the secret.

`versionId` (Python: `version_id`)

The ID of this version of the secret.

`versionStage` (Python: `version_stage`)

The list of staging labels attached to this version of the secret.

`secretValue` (Python: `secret_value`)

The value of this version of the secret. This object, `SecretValue`, contains the following information.

`secretString` (Python: `secret_string`)

The decrypted part of the protected secret information that you provided to Secrets Manager as a string.

`secretBinary` (Python: `secret_binary`)

(Optional) The decrypted part of the protected secret information that you provided to Secrets Manager as binary data in the form of a byte array. This property contains the binary data as a base64-encoded string.

This property isn't used if you created the secret in the Secrets Manager console.

Examples

The following examples demonstrate how to call this operation in custom component code.

Java (IPC client V1)

Example: Get a secret value

> **Note**
>
> This example uses an `IPCUtils` class to create a connection to the AWS IoT Greengrass Core IPC service. For more information, see Connect to the AWS IoT Greengrass Core IPC service.

```
package com.aws.greengrass.docs.samples.ipc;

import com.aws.greengrass.docs.samples.ipc.util.IPCUtils;
import software.amazon.awssdk.aws.greengrass.GetSecretValueResponseHandler;
import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClient;
import software.amazon.awssdk.aws.greengrass.model.GetSecretValueRequest;
import software.amazon.awssdk.aws.greengrass.model.GetSecretValueResponse;
import software.amazon.awssdk.aws.greengrass.model.UnauthorizedError;
import software.amazon.awssdk.eventstreamrpc.EventStreamRPCConnection;
```

```
import java.util.Optional;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public class GetSecretValue {

    public static final int TIMEOUT_SECONDS = 10;

    public static void main(String[] args) {
        String secretArn = args[0];
        String versionStage = args[1];
        try (EventStreamRPCConnection eventStreamRPCConnection =
                     IPCUtils.getEventStreamRpcConnection()) {
            GreengrassCoreIPCClient ipcClient =
                    new GreengrassCoreIPCClient(eventStreamRPCConnection);
            GetSecretValueResponseHandler responseHandler =
                    GetSecretValue.getSecretValue(ipcClient, secretArn,
 versionStage);
            CompletableFuture<GetSecretValueResponse> futureResponse =
                    responseHandler.getResponse();
            try {
                GetSecretValueResponse response =
 futureResponse.get(TIMEOUT_SECONDS, TimeUnit.SECONDS);
                response.getSecretValue().postFromJson();
                String secretString = response.getSecretValue().getSecretString();
                System.out.println("Successfully retrieved secret value: " +
 secretString);
            } catch (TimeoutException e) {
                System.err.println("Timeout occurred while retrieving secret: " +
 secretArn);
            } catch (ExecutionException e) {
                if (e.getCause() instanceof UnauthorizedError) {
                    System.err.println("Unauthorized error while retrieving secret:
 " + secretArn);
                } else {
                    throw e;
                }
            }
        } catch (InterruptedException e) {
            System.out.println("IPC interrupted.");
        } catch (ExecutionException e) {
```

```
            System.err.println("Exception occurred when using IPC.");
            e.printStackTrace();
            System.exit(1);
        }
    }

    public static GetSecretValueResponseHandler
 getSecretValue(GreengrassCoreIPCClient greengrassCoreIPCClient, String secretArn,
 String versionStage) {
        GetSecretValueRequest getSecretValueRequest = new GetSecretValueRequest();
        getSecretValueRequest.setSecretId(secretArn);
        getSecretValueRequest.setVersionStage(versionStage);
        return greengrassCoreIPCClient.getSecretValue(getSecretValueRequest,
 Optional.empty());
    }
}
```

Python (IPC client V1)

Example: Get a secret value

> **Note**
>
> This example assumes that you are using version 1.5.4 or later of the AWS IoT Device SDK for Python v2.

```
import json

import awsiot.greengrasscoreipc
from awsiot.greengrasscoreipc.model import (
    GetSecretValueRequest,
    GetSecretValueResponse,
    UnauthorizedError
)

secret_id = 'arn:aws:secretsmanager:us-
west-2:123456789012:secret:MyGreengrassSecret-abcdef'
TIMEOUT = 10

ipc_client = awsiot.greengrasscoreipc.connect()

request = GetSecretValueRequest()
```

```
request.secret_id = secret_id
request.version_stage = 'AWSCURRENT'
operation = ipc_client.new_get_secret_value()
operation.activate(request)
future_response = operation.get_response()
response = future_response.result(TIMEOUT)
secret_json = json.loads(response.secret_value.secret_string)
# Handle secret value.
```

JavaScript

Example: Get a secret value

```
import {
    GetSecretValueRequest,
} from 'aws-iot-device-sdk-v2/dist/greengrasscoreipc/model';
import * as greengrasscoreipc from "aws-iot-device-sdk-v2/dist/greengrasscoreipc";

class GetSecretValue {
    private readonly secretId : string;
    private readonly versionStage : string;
    private ipcClient : greengrasscoreipc.Client

    constructor() {
        this.secretId = "<define_your_own_secretId>"
        this.versionStage = "<define_your_own_versionStage>"

        this.getSecretValue().then(r => console.log("Started workflow"));
    }

    private async getSecretValue() {
        try {
            this.ipcClient = await getIpcClient();

            const getSecretValueRequest : GetSecretValueRequest = {
                secretId: this.secretId,
                versionStage: this.versionStage,
            };

            const result = await
 this.ipcClient.getSecretValue(getSecretValueRequest);
            const secretString = result.secretValue.secretString;
            console.log("Successfully retrieved secret value: " + secretString)
```

```
        } catch (e) {
            // parse the error depending on your use cases
            throw e
        }
    }
}

export async function getIpcClient(){
    try {
        const ipcClient = greengrasscoreipc.createClient();
        await ipcClient.connect()
            .catch(error => {
                // parse the error depending on your use cases
                throw error;
            });
        return ipcClient
    } catch (err) {
        // parse the error depending on your use cases
        throw err
    }
}

const getSecretValue = new GetSecretValue();
```

Examples

Use the following examples to learn how to use the secret manager IPC service in your components.

Example: Print secret (Python, IPC client V1)

This example component prints the value of a secret that you deploy to the core device.

> **Important**
>
> This example component prints the value of a secret, so use it only with secrets that store test data. Don't use this component to print the value of a secret that stores important information.

Topics

- Recipe
- Artifacts
- Usage

Recipe

The following example recipe defines a secret ARN configuration parameter and allows the component to get the value of any secret on the core device.

> **Note**
>
> We recommend that in a production environment, you reduce the scope of the authorization policy, so that the component retrieves only the secrets that it uses. You can change the * wildcard to a list of secret ARNs when you deploy the component.

JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.PrintSecret",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "Prints the value of an AWS Secrets Manager secret.",
  "ComponentPublisher": "Amazon",
  "ComponentDependencies": {
    "aws.greengrass.SecretManager": {
      "VersionRequirement": "^2.0.0",
      "DependencyType": "HARD"
    }
  },
  "ComponentConfiguration": {
    "DefaultConfiguration": {
      "SecretArn": "",
      "accessControl": {
        "aws.greengrass.SecretManager": {
          "com.example.PrintSecret:secrets:1": {
            "policyDescription": "Allows access to a secret.",
            "operations": [
              "aws.greengrass#GetSecretValue"
            ],
            "resources": [
```

```
              "*"
            ]
          }
        }
      }
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "install": "python3 -m pip install --user awsiotsdk",
        "Run": "python3 -u {artifacts:path}/print_secret.py \"{configuration:/
SecretArn}\""
      }
    },
    {
      "Platform": {
        "os": "windows"
      },
      "Lifecycle": {
        "install": "py -3 -m pip install --user awsiotsdk",
        "Run": "py -3 -u {artifacts:path}/print_secret.py \"{configuration:/
SecretArn}\""
      }
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.PrintSecret
ComponentVersion: 1.0.0
ComponentDescription: Prints the value of a Secrets Manager secret.
ComponentPublisher: Amazon
ComponentDependencies:
  aws.greengrass.SecretManager:
    VersionRequirement: "^2.0.0"
    DependencyType: HARD
```

```
ComponentConfiguration:
  DefaultConfiguration:
    SecretArn: ''
    accessControl:
      aws.greengrass.SecretManager:
        com.example.PrintSecret:secrets:1:
          policyDescription: Allows access to a secret.
          operations:
            - aws.greengrass#GetSecretValue
          resources:
            - "*"
Manifests:
  - Platform:
      os: linux
    Lifecycle:
      install: python3 -m pip install --user awsiotsdk
      Run: python3 -u {artifacts:path}/print_secret.py "{configuration:/SecretArn}"
  - Platform:
      os: windows
    Lifecycle:
      install: py -3 -m pip install --user awsiotsdk
      Run: py -3 -u {artifacts:path}/print_secret.py "{configuration:/SecretArn}"
```

Artifacts

The following example Python application demonstrates how to use the secret manager IPC service to get the value of a secret on the core device.

```
import concurrent.futures
import json
import sys
import traceback

import awsiot.greengrasscoreipc
from awsiot.greengrasscoreipc.model import (
    GetSecretValueRequest,
    GetSecretValueResponse,
    UnauthorizedError
)

TIMEOUT = 10
```

```
if len(sys.argv) == 1:
    print('Provide SecretArn in the component configuration.', file=sys.stdout)
    exit(1)

secret_id = sys.argv[1]

try:
    ipc_client = awsiot.greengrasscoreipc.connect()

    request = GetSecretValueRequest()
    request.secret_id = secret_id
    operation = ipc_client.new_get_secret_value()
    operation.activate(request)
    future_response = operation.get_response()

    try:
        response = future_response.result(TIMEOUT)
        secret_json = json.loads(response.secret_value.secret_string)
        print('Successfully got secret: ' + secret_id)
        print('Secret value: ' + str(secret_json))
    except concurrent.futures.TimeoutError:
        print('Timeout occurred while getting secret: ' + secret_id, file=sys.stderr)
    except UnauthorizedError as e:
        print('Unauthorized error while getting secret: ' + secret_id,
 file=sys.stderr)
        raise e
    except Exception as e:
        print('Exception while getting secret: ' + secret_id, file=sys.stderr)
        raise e
except Exception:
    print('Exception occurred when using IPC.', file=sys.stderr)
    traceback.print_exc()
    exit(1)
```

Usage

You can use this example component with the secret manager component to deploy and print the value of a secret on your core device.

To create, deploy, and print a test secret

1. Create a Secrets Manager secret with test data.

Linux or Unix

```
aws secretsmanager create-secret \
  --name MyTestGreengrassSecret \
  --secret-string '{"my-secret-key": "my-secret-value"}'
```

Windows Command Prompt (CMD)

```
aws secretsmanager create-secret ^
  --name MyTestGreengrassSecret ^
  --secret-string '{"my-secret-key": "my-secret-value"}'
```

PowerShell

```
aws secretsmanager create-secret `
  --name MyTestGreengrassSecret `
  --secret-string '{"my-secret-key": "my-secret-value"}'
```

Save the ARN of the secret to use in the following steps.

For more information, see Creating a secret in the *AWS Secrets Manager User Guide*.

2. Deploy the secret manager component (`aws.greengrass.SecretManager`) with the following configuration merge update. Specify the ARN of the secret that you created earlier.

```
{
  "cloudSecrets": [
    {
      "arn": "arn:aws:secretsmanager:us-
west-2:123456789012:secret:MyTestGreengrassSecret-abcdef"
    }
  ]
}
```

For more information, see Deploy AWS IoT Greengrass components to devices or the Greengrass CLI deployment command.

3. Create and deploy the example component in this section with the following configuration merge update. Specify the ARN of the secret that you created earlier.

```
{
  "SecretArn": "arn:aws:secretsmanager:us-
west-2:123456789012:secret:MyTestGreengrassSecret",
  "accessControl": {
    "aws.greengrass.SecretManager": {
      "com.example.PrintSecret:secrets:1": {
        "policyDescription": "Allows access to a secret.",
        "operations": [
          "aws.greengrass#GetSecretValue"
        ],
        "resources": [
          "arn:aws:secretsmanager:us-
west-2:123456789012:secret:MyTestGreengrassSecret-abcdef"
        ]
      }
    }
  }
}
```

For more information, see Create AWS IoT Greengrass components

4. View the AWS IoT Greengrass Core software logs to verify that the deployments succeed, and view the `com.example.PrintSecret` component log to see the secret value printed. For more information, see Monitor AWS IoT Greengrass logs.

Interact with local shadows

Use the shadow IPC service to interact with local shadows on a device. The device you choose to interact with can be your core device or a connected client device.

To use these IPC operations, include the shadow manager component as a dependency in your custom component. You can then use IPC operations in your custom components to interact with local shadows on your device through the shadow manager. To enable custom components to react to changes in local shadow states, you can also use the publish/subscribe IPC service to subscribe to shadow events. For more information about using the publish/subscribe service, see the Publish/subscribe local messages.

> **Note**
>
> To enable a core device to interact with client device shadows, you must also configure and deploy the MQTT bridge component. For more information, see Enable shadow manager to communicate with client devices.

Topics

- Minimum SDK versions
- Authorization
- GetThingShadow
- UpdateThingShadow
- DeleteThingShadow
- ListNamedShadowsForThing

Minimum SDK versions

The following table lists the minimum versions of the SDKs that you can use to interact with local shadows.

SDK	Minimum version	
AWS IoT Greengrass Component SDK (C, C++, Rust)	v1.0.0	
AWS IoT Device SDK for Java v2	v1.4.0	
AWS IoT Device SDK for Python v2	v1.6.0	
AWS IoT Device SDK for C++ v2	v1.17.0	

SDK	Minimum version	
AWS IoT Device SDK for JavaScript v2	v1.12.0	

Authorization

To use the shadow IPC service in a custom component, you must define authorization policies that allow your component to interact with shadows. For information about defining authorization policies, see Authorize components to perform IPC operations.

Authorization policies for shadow interaction have the following properties.

IPC service identifier: `aws.greengrass.ShadowManager`

Operation	Description	Resources
`aws.greengrass#GetThingShadow`	Allows a component to retrieve the shadow of a thing.	One of the following strings: • `$aws/things/` *`thingName`* `/shadow/`, to allow access to the classic device shadow. • `$aws/things/` *`thingName`* `/shadow/name/` *`shadowName`* , to allow access to a named shadow. • `*` to allow access to all shadows.
`aws.greengrass#UpdateThingShadow`	Allows a component to update the shadow of a thing.	One of the following strings: • `$aws/things/` *`thingName`* `/`

Operation	Description	Resources
		`shadow/`, to allow access to the classic device shadow. • `$aws/things/` *thingName* `/shadow/name/` *shadowName* , to allow access to a named shadow. • `*` to allow access to all shadows.
`aws.greengrass#DeleteThingShadow`	Allows a component to delete the shadow of a thing.	One of the following strings: • `$aws/things/` *thingName* `/shadow/`, to allow access to the classic device shadow • `$aws/things/` *thingName* `/shadow/name/` *shadowName* , to allow access to a named shadow • `*`, to allow access to all shadows.
`aws.greengrass#ListNamedShadowsForThing`	Allows a component to retrieve the list of named shadows for a thing.	A thing name string that allows access to the thing to list its shadows. Use `*` to allow access to all things.

IPC service identifier: `aws.greengrass.ipc.pubsub`

Operation	Description	Resources
`aws.greengrass#SubscribeToTopic`	Allows a component to subscribe to messages for the topics that you specify.	One of the following topic strings: • *shadowTopicPrefix* / get/accepted • *shadowTopicPrefix* / get/rejected • *shadowTopicPrefix* / delete/accepted • *shadowTopicPrefix* / delete/rejected • *shadowTopicPrefix* / update/accepted • *shadowTopicPrefix* / update/delta • *shadowTopicPrefix* / update/rejected The value of the topic prefix *shadowTopicPrefix* depends on the type of shadow: • Classic shadow: $aws/things/ *thingName* /shadow • Named shadow: $aws/things/ *thingName* /shadow/name/ *shadowName* Use * to allow access to all topics.

Operation	Description	Resources
		In Greengrass nucleus v2.6.0 and later, you can subscribe to topics that contain MQTT topic wildcards (# and +). This topic string supports MQTT topic wildcards as literal characters. For example, if a component's authorization policy grants access to `test/topic/#` , the component can subscribe to `test/topic/#` , but it can't subscribe to `test/topic/filter` .

Recipe variables in local shadow authorization policies

If you use v2.6.0 or later of the Greengrass nucleus, and you set the Greengrass nucleus' interpolateComponentConfiguration configuration option to `true`, you can use the `{iot:thingName}` recipe variable in authorization policies. This feature enables you to configure a single authorization policy for a group of core devices, where each core device can access only its own shadow. For example, you can allow a component access to the following resource for shadow IPC operations.

```
$aws/things/{iot:thingName}/shadow/
```

Authorization policy examples

You can reference the following authorization policy examples to help you configure authorization policies for your components.

Example: Allow a group of core devices to interact with local shadows

> **Important**
>
> This example uses a feature that is available for v2.6.0 and later of the Greengrass nucleus component. Greengrass nucleus v2.6.0 adds support for most recipe variables, such

as `{iot:thingName}`, in component configurations. To enable this feature, set the Greengrass nucleus' interpolateComponentConfiguration configuration option to `true`. For an example that works for all versions of the Greengrass nucleus, see the example authorization policy for a single core device.

The following example authorization policy allows the component `com.example.MyShadowInteractionComponent` to interact with the classic device shadow and the named shadow `myNamedShadow` for the core device that runs the component. This policy also allows this component to receive messages on local topics for these shadows.

JSON

```
{
  "accessControl": {
    "aws.greengrass.ShadowManager": {
      "com.example.MyShadowInteractionComponent:shadow:1": {
        "policyDescription": "Allows access to shadows",
        "operations": [
          "aws.greengrass#GetThingShadow",
          "aws.greengrass#UpdateThingShadow",
          "aws.greengrass#DeleteThingShadow"
        ],
        "resources": [
          "$aws/things/{iot:thingName}/shadow",
          "$aws/things/{iot:thingName}/shadow/name/myNamedShadow"
        ]
      },
      "com.example.MyShadowInteractionComponent:shadow:2": {
        "policyDescription": "Allows access to things with shadows",
        "operations": [
          "aws.greengrass#ListNamedShadowsForThing"
        ],
        "resources": [
          "{iot:thingName}"
        ]
      }
    },
    "aws.greengrass.ipc.pubsub": {
      "com.example.MyShadowInteractionComponent:pubsub:1": {
        "policyDescription": "Allows access to shadow pubsub topics",
        "operations": [
```

```
          "aws.greengrass#SubscribeToTopic"
        ],
        "resources": [
          "$aws/things/{iot:thingName}/shadow/get/accepted",
          "$aws/things/{iot:thingName}/shadow/name/myNamedShadow/get/accepted"
        ]
      }
    }
  }
}
```

YAML

```
accessControl:
  aws.greengrass.ShadowManager:
    'com.example.MyShadowInteractionComponent:shadow:1':
      policyDescription: 'Allows access to shadows'
      operations:
        - 'aws.greengrass#GetThingShadow'
        - 'aws.greengrass#UpdateThingShadow'
        - 'aws.greengrass#DeleteThingShadow'
      resources:
        - $aws/things/{iot:thingName}/shadow
        - $aws/things/{iot:thingName}/shadow/name/myNamedShadow
    'com.example.MyShadowInteractionComponent:shadow:2':
      policyDescription: 'Allows access to things with shadows'
      operations:
        - 'aws.greengrass#ListNamedShadowsForThing'
      resources:
        - '{iot:thingName}'
  aws.greengrass.ipc.pubsub:
    'com.example.MyShadowInteractionComponent:pubsub:1':
      policyDescription: 'Allows access to shadow pubsub topics'
      operations:
        - 'aws.greengrass#SubscribeToTopic'
      resources:
        - $aws/things/{iot:thingName}/shadow/get/accepted
        - $aws/things/{iot:thingName}/shadow/name/myNamedShadow/get/accepted
```

Example: Allow a group of core devices to interact with client device shadows

> **Important**
>
> This feature requires Greengrass nucleus v2.6.0 or later, shadow manager v2.2.0 or later, and MQTT bridge v2.2.0 or later. You must configure MQTT bridge to enable shadow manager to communicate with client devices.

The following example authorization policy allows the component `com.example.MyShadowInteractionComponent` to interact with all device shadows for client devices whose names start with `MyClientDevice`.

> **Note**
>
> To enable a core device to interact with client device shadows, you must also configure and deploy the MQTT bridge component. For more information, see Enable shadow manager to communicate with client devices.

JSON

```
{
  "accessControl": {
    "aws.greengrass.ShadowManager": {
      "com.example.MyShadowInteractionComponent:shadow:1": {
        "policyDescription": "Allows access to shadows",
        "operations": [
          "aws.greengrass#GetThingShadow",
          "aws.greengrass#UpdateThingShadow",
          "aws.greengrass#DeleteThingShadow"
        ],
        "resources": [
          "$aws/things/MyClientDevice*/shadow",
          "$aws/things/MyClientDevice*/shadow/name/*"
        ]
      },
      "com.example.MyShadowInteractionComponent:shadow:2": {
        "policyDescription": "Allows access to things with shadows",
        "operations": [
          "aws.greengrass#ListNamedShadowsForThing"
```

```
        ],
        "resources": [
          "MyClientDevice*"
        ]
      }
    }
  }
}
```

YAML

```
accessControl:
  aws.greengrass.ShadowManager:
    'com.example.MyShadowInteractionComponent:shadow:1':
      policyDescription: 'Allows access to shadows'
      operations:
        - 'aws.greengrass#GetThingShadow'
        - 'aws.greengrass#UpdateThingShadow'
        - 'aws.greengrass#DeleteThingShadow'
      resources:
        - $aws/things/MyClientDevice*/shadow
        - $aws/things/MyClientDevice*/shadow/name/*
    'com.example.MyShadowInteractionComponent:shadow:2':
      policyDescription: 'Allows access to things with shadows'
      operations:
        - 'aws.greengrass#ListNamedShadowsForThing'
      resources:
        - MyClientDevice*
```

Example: Allow a single core device to interact with local shadows

The following example authorization policy allows the component `com.example.MyShadowInteractionComponent` to interact with the classic device shadow and the named shadow `myNamedShadow` for the device `MyThingName`. This policy also allows this component to receive messages on local topics for these shadows.

JSON

```
{
  "accessControl": {
    "aws.greengrass.ShadowManager": {
```

```
      "com.example.MyShadowInteractionComponent:shadow:1": {
        "policyDescription": "Allows access to shadows",
        "operations": [
          "aws.greengrass#GetThingShadow",
          "aws.greengrass#UpdateThingShadow",
          "aws.greengrass#DeleteThingShadow"
        ],
        "resources": [
          "$aws/things/MyThingName/shadow",
          "$aws/things/MyThingName/shadow/name/myNamedShadow"
        ]
      },
      "com.example.MyShadowInteractionComponent:shadow:2": {
        "policyDescription": "Allows access to things with shadows",
        "operations": [
          "aws.greengrass#ListNamedShadowsForThing"
        ],
        "resources": [
          "MyThingName"
        ]
      }
    },
    "aws.greengrass.ipc.pubsub": {
      "com.example.MyShadowInteractionComponent:pubsub:1": {
        "policyDescription": "Allows access to shadow pubsub topics",
        "operations": [
          "aws.greengrass#SubscribeToTopic"
        ],
        "resources": [
          "$aws/things/MyThingName/shadow/get/accepted",
          "$aws/things/MyThingName/shadow/name/myNamedShadow/get/accepted"
        ]
      }
    }
  }
}
```

YAML

```
accessControl:
  aws.greengrass.ShadowManager:
    'com.example.MyShadowInteractionComponent:shadow:1':
      policyDescription: 'Allows access to shadows'
```

```
      operations:
        - 'aws.greengrass#GetThingShadow'
        - 'aws.greengrass#UpdateThingShadow'
        - 'aws.greengrass#DeleteThingShadow'
      resources:
        - $aws/things/MyThingName/shadow
        - $aws/things/MyThingName/shadow/name/myNamedShadow
    'com.example.MyShadowInteractionComponent:shadow:2':
      policyDescription: 'Allows access to things with shadows'
      operations:
        - 'aws.greengrass#ListNamedShadowsForThing'
      resources:
        - MyThingName
  aws.greengrass.ipc.pubsub:
    'com.example.MyShadowInteractionComponent:pubsub:1':
      policyDescription: 'Allows access to shadow pubsub topics'
      operations:
        - 'aws.greengrass#SubscribeToTopic'
      resources:
        - $aws/things/MyThingName/shadow/get/accepted
        - $aws/things/MyThingName/shadow/name/myNamedShadow/get/accepted
```

Example: Allow a group of core devices to react to local shadow state changes

> **Important**
>
> This example uses a feature that is available for v2.6.0 and later of the Greengrass nucleus component. Greengrass nucleus v2.6.0 adds support for most recipe variables, such as `{iot:thingName}`, in component configurations. To enable this feature, set the Greengrass nucleus' interpolateComponentConfiguration configuration option to `true`. For an example that works for all versions of the Greengrass nucleus, see the example authorization policy for a single core device.

The following example access control policy allows the custom `com.example.MyShadowReactiveComponent` to receive messages on the `/update/delta` topic for the classic device shadow and the named shadow `myNamedShadow` on each core device that runs the component.

JSON

```
{
  "accessControl": {
    "aws.greengrass.ipc.pubsub": {
      "com.example.MyShadowReactiveComponent:pubsub:1": {
        "policyDescription": "Allows access to shadow pubsub topics",
        "operations": [
          "aws.greengrass#SubscribeToTopic"
        ],
        "resources": [
          "$aws/things/{iot:thingName}/shadow/update/delta",
          "$aws/things/{iot:thingName}/shadow/name/myNamedShadow/update/delta"
        ]
      }
    }
  }
}
```

YAML

```
accessControl:
  aws.greengrass.ipc.pubsub:
    "com.example.MyShadowReactiveComponent:pubsub:1":
      policyDescription: Allows access to shadow pubsub topics
      operations:
        - 'aws.greengrass#SubscribeToTopic'
      resources:
        - $aws/things/{iot:thingName}/shadow/update/delta
        - $aws/things/{iot:thingName}/shadow/name/myNamedShadow/update/delta
```

Example: Allow a single core device to react to local shadow state changes

The following example access control policy allows the custom `com.example.MyShadowReactiveComponent` to receive messages on the `/update/delta` topic for the classic device shadow and the named shadow `myNamedShadow` for the device `MyThingName`.

JSON

```
{
```

```
  "accessControl": {
    "aws.greengrass.ipc.pubsub": {
      "com.example.MyShadowReactiveComponent:pubsub:1": {
        "policyDescription": "Allows access to shadow pubsub topics",
        "operations": [
          "aws.greengrass#SubscribeToTopic"
        ],
        "resources": [
          "$aws/things/MyThingName/shadow/update/delta",
          "$aws/things/MyThingName/shadow/name/myNamedShadow/update/delta"
        ]
      }
    }
  }
}
```

YAML

```
accessControl:
  aws.greengrass.ipc.pubsub:
    "com.example.MyShadowReactiveComponent:pubsub:1":
      policyDescription: Allows access to shadow pubsub topics
      operations:
        - 'aws.greengrass#SubscribeToTopic'
      resources:
        - $aws/things/MyThingName/shadow/update/delta
        - $aws/things/MyThingName/shadow/name/myNamedShadow/update/delta
```

GetThingShadow

Get the shadow for a specified thing.

Request

This operation's request has the following parameters:

`thingName` (Python: `thing_name`)

The name of the thing.

Type: `string`

`shadowName` (Python: `shadow_name`)

The name of the shadow. To specify the thing's classic shadow, set this parameter to an empty string (`""`).

> **Warning**
>
> The AWS IoT Greengrass service uses the `AWSManagedGreengrassV2Deployment` named shadow to manage deployments that target individual core devices. This named shadow is reserved for use by the AWS IoT Greengrass service. Do not update or delete this named shadow.

Type: `string`

Response

This operation's response has the following information:

`payload`

The response state document as a blob.

Type: `object` that contains the following information:

`state`

The state information.

This object contains the following information.

`desired`

The state properties and values requested to be updated in the device.

Type: `map` of key-value pairs

`reported`

The state properties and values reported by the device.

Type: `map` of key-value pairs

`delta`

The difference between the desired and reported state properties and values. This property is present only if the `desired` and `reported` states are different.

Type: `map` of key-value pairs

`metadata`

The timestamps for each attribute in the `desired` and `reported` sections so that you can determine when the state was updated.

Type: `string`

`timestamp`

The epoch date and time that the response was generated.

Type: `integer`

`clientToken` (Python: `clientToken`)

The token that is used to match the request and corresponding response

Type: `string`

`version`

The version of the local shadow document.

Type: `integer`

Errors

This operation can return the following errors.

`InvalidArgumentsError`

The local shadow service is unable to validate the request parameters. This can occur if the request contains malformed JSON or unsupported characters.

`ResourceNotFoundError`

The requested local shadow document can't be found.

`ServiceError`

An internal service error occurred, or the number of requests to the IPC service exceeded the limits specified in the `maxLocalRequestsPerSecondPerThing` and `maxTotalLocalRequestsRate` configuration parameters in the shadow manager component.

`UnauthorizedError`

The component's authorization policy doesn't include required permissions for this operation.

Examples

The following examples demonstrate how to call this operation in custom component code.

Java (IPC client V1)

Example: Get a thing shadow

> **Note**
>
> This example uses an `IPCUtils` class to create a connection to the AWS IoT Greengrass Core IPC service. For more information, see Connect to the AWS IoT Greengrass Core IPC service.

```
package com.aws.greengrass.docs.samples.ipc;

import com.aws.greengrass.docs.samples.ipc.util.IPCUtils;
import software.amazon.awssdk.aws.greengrass.GetThingShadowResponseHandler;
import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClient;
import software.amazon.awssdk.aws.greengrass.model.GetThingShadowRequest;
import software.amazon.awssdk.aws.greengrass.model.GetThingShadowResponse;
import software.amazon.awssdk.aws.greengrass.model.ResourceNotFoundError;
import software.amazon.awssdk.aws.greengrass.model.UnauthorizedError;
import software.amazon.awssdk.eventstreamrpc.EventStreamRPCConnection;

import java.nio.charset.StandardCharsets;
import java.util.Optional;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;
```

```
public class GetThingShadow {

    public static final int TIMEOUT_SECONDS = 10;

    public static void main(String[] args) {
        // Use the current core device's name if thing name isn't set.
        String thingName = args[0].isEmpty() ? System.getenv("AWS_IOT_THING_NAME") :
 args[0];
        String shadowName = args[1];
        try (EventStreamRPCConnection eventStreamRPCConnection =
                     IPCUtils.getEventStreamRpcConnection()) {
            GreengrassCoreIPCClient ipcClient =
                    new GreengrassCoreIPCClient(eventStreamRPCConnection);
            GetThingShadowResponseHandler responseHandler =
                    GetThingShadow.getThingShadow(ipcClient, thingName,
 shadowName);
            CompletableFuture<GetThingShadowResponse> futureResponse =
                    responseHandler.getResponse();
            try {
                GetThingShadowResponse response =
 futureResponse.get(TIMEOUT_SECONDS,
                        TimeUnit.SECONDS);
                String shadowPayload = new String(response.getPayload(),
 StandardCharsets.UTF_8);
                System.out.printf("Successfully got shadow %s/%s: %s%n", thingName,
 shadowName,
                        shadowPayload);
            } catch (TimeoutException e) {
                System.err.printf("Timeout occurred while getting shadow: %s/%s%n",
 thingName,
                        shadowName);
            } catch (ExecutionException e) {
                if (e.getCause() instanceof UnauthorizedError) {
                    System.err.printf("Unauthorized error while getting shadow: %s/
%s%n",
                            thingName, shadowName);
                } else if (e.getCause() instanceof ResourceNotFoundError) {
                    System.err.printf("Unable to find shadow to get: %s/%s%n",
 thingName,
                            shadowName);
                } else {
                    throw e;
                }
```

```
            }
        } catch (InterruptedException e) {
            System.out.println("IPC interrupted.");
        } catch (ExecutionException e) {
            System.err.println("Exception occurred when using IPC.");
            e.printStackTrace();
            System.exit(1);
        }
    }

    public static GetThingShadowResponseHandler
 getThingShadow(GreengrassCoreIPCClient greengrassCoreIPCClient, String thingName,
 String shadowName) {
        GetThingShadowRequest getThingShadowRequest = new GetThingShadowRequest();
        getThingShadowRequest.setThingName(thingName);
        getThingShadowRequest.setShadowName(shadowName);
        return greengrassCoreIPCClient.getThingShadow(getThingShadowRequest,
 Optional.empty());
    }
}
```

Python (IPC client V1)

Example: Get a thing shadow

```
import awsiot.greengrasscoreipc
import awsiot.greengrasscoreipc.client as client
from awsiot.greengrasscoreipc.model import GetThingShadowRequest

TIMEOUT = 10

def sample_get_thing_shadow_request(thingName, shadowName):
    try:
        # set up IPC client to connect to the IPC server
        ipc_client = awsiot.greengrasscoreipc.connect()

        # create the GetThingShadow request
        get_thing_shadow_request = GetThingShadowRequest()
        get_thing_shadow_request.thing_name = thingName
        get_thing_shadow_request.shadow_name = shadowName

        # retrieve the GetThingShadow response after sending the request to the IPC
 server
```

```
        op = ipc_client.new_get_thing_shadow()
        op.activate(get_thing_shadow_request)
        fut = op.get_response()

        result = fut.result(TIMEOUT)
        return result.payload

    except InvalidArgumentsError as e:
        # add error handling
        ...
    # except ResourceNotFoundError | UnauthorizedError | ServiceError
```

JavaScript

Example: Get a thing shadow

```
import {
    GetThingShadowRequest
} from 'aws-iot-device-sdk-v2/dist/greengrasscoreipc/model';
import * as greengrasscoreipc from 'aws-iot-device-sdk-v2/dist/greengrasscoreipc';

class GetThingShadow {
    private ipcClient: greengrasscoreipc.Client;
    private thingName: string;
    private shadowName: string;

    constructor() {
        // Define args parameters here
        this.thingName = "<define_your_own_thingName>";
        this.shadowName = "<define_your_own_shadowName>";
        this.bootstrap();
    }

    async bootstrap() {
        try {
            this.ipcClient = await getIpcClient();
        } catch (err) {
            // parse the error depending on your use cases
            throw err
        }

        try {
            await this.handleGetThingShadowOperation(this.thingName,
                this.shadowName);
```

```
        } catch (err) {
            // parse the error depending on your use cases
            throw err
        }
    }

    async handleGetThingShadowOperation(
        thingName: string,
        shadowName: string
    ) {
        const request: GetThingShadowRequest = {
            thingName: thingName,
            shadowName: shadowName
        };
        const response = await this.ipcClient.getThingShadow(request);
    }
}

export async function getIpcClient() {
    try {
        const ipcClient = greengrasscoreipc.createClient();
        await ipcClient.connect()
            .catch(error => {
                // parse the error depending on your use cases
                throw error;
            });
        return ipcClient
    } catch (err) {
        // parse the error depending on your use caseså
        throw err
    }
}

const startScript = new GetThingShadow();
```

UpdateThingShadow

Update the shadow for the specified thing. If a shadow doesn't exist, one is created.

Request

This operation's request has the following parameters:

`thingName` (Python: `thing_name`)

The name of the thing.

Type: `string`

`shadowName` (Python: `shadow_name`)

The name of the shadow. To specify the thing's classic shadow, set this parameter to an empty string (`""`).

> **Warning**
>
> The AWS IoT Greengrass service uses the `AWSManagedGreengrassV2Deployment` named shadow to manage deployments that target individual core devices. This named shadow is reserved for use by the AWS IoT Greengrass service. Do not update or delete this named shadow.

Type: `string`

`payload`

The request state document as a blob.

Type: `object` that contains the following information:

`state`

The state information to update. This IPC operation affects only the specified fields.

This object contains the following information. Typically, you'll use either the `desired` property or the `reported` property, but not both in the same request.

`desired`

The state properties and values requested to be updated in the device.

Type: `map` of key-value pairs

`reported`

The state properties and values reported by the device.

Type: `map` of key-value pairs

`clientToken` (Python: `client_token`)

(Optional) The token that is used to match the request and corresponding response by the client token.

Type: `string`

`version`

(Optional) The version of the local shadow document to update. The shadow service processes the update only if the specified version matches the latest version that it has.

Type: `integer`

Response

This operation's response has the following information:

`payload`

The response state document as a blob.

Type: `object` that contains the following information:

`state`

The state information.

This object contains the following information.

`desired`

The state properties and values requested to be updated in the device.

Type: `map` of key-value pairs

`reported`

The state properties and values reported by the device.

Type: `map` of key-value pairs

`delta`

The state properties and values reported by the device.

Type: `map` of key-value pairs

`metadata`

The timestamps for each attribute in the `desired` and `reported` sections so that you can determine when the state was updated.

Type: `string`

`timestamp`

The epoch date and time that the response was generated.

Type: `integer`

`clientToken` (Python: `client_token`)

The token that is used to match the request and corresponding response.

Type: `string`

`version`

The version of local shadow document after the update is complete.

Type: `integer`

Errors

This operation can return the following errors.

`ConflictError`

The local shadow service encountered a version conflict during the update operation. This occurs when the version in the request payload doesn't match the version in the latest available local shadow document.

`InvalidArgumentsError`

The local shadow service is unable to validate the request parameters. This can occur if the request contains malformed JSON or unsupported characters.

A valid `payload` has the following properties:

- The `state` node exists, and is an object that contains the `desired` or `reported` state information.
- The `desired` and `reported` nodes are either objects or null. At least one of these objects must contain valid state information.

- The depth of the `desired` and `reported` objects can't exceed eight nodes.
- The length of the `clientToken` value can't exceed 64 characters.
- The `version` value must be 1 or higher.

`ServiceError`

An internal service error occurred, or the number of requests to the IPC service exceeded the limits specified in the `maxLocalRequestsPerSecondPerThing` and `maxTotalLocalRequestsRate` configuration parameters in the shadow manager component.

`UnauthorizedError`

The component's authorization policy doesn't include required permissions for this operation.

Examples

The following examples demonstrate how to call this operation in custom component code.

Java (IPC client V1)

Example: Update a thing shadow

> **Note**
>
> This example uses an `IPCUtils` class to create a connection to the AWS IoT Greengrass Core IPC service. For more information, see Connect to the AWS IoT Greengrass Core IPC service.

```
package com.aws.greengrass.docs.samples.ipc;

import com.aws.greengrass.docs.samples.ipc.util.IPCUtils;
import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClient;
import software.amazon.awssdk.aws.greengrass.UpdateThingShadowResponseHandler;
import software.amazon.awssdk.aws.greengrass.model.UnauthorizedError;
import software.amazon.awssdk.aws.greengrass.model.UpdateThingShadowRequest;
import software.amazon.awssdk.aws.greengrass.model.UpdateThingShadowResponse;
import software.amazon.awssdk.eventstreamrpc.EventStreamRPCConnection;

import java.nio.charset.StandardCharsets;
import java.util.Optional;
import java.util.concurrent.CompletableFuture;
```

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public class UpdateThingShadow {

    public static final int TIMEOUT_SECONDS = 10;

    public static void main(String[] args) {
        // Use the current core device's name if thing name isn't set.
        String thingName = args[0].isEmpty() ? System.getenv("AWS_IOT_THING_NAME") :
 args[0];
        String shadowName = args[1];
        byte[] shadowPayload = args[2].getBytes(StandardCharsets.UTF_8);
        try (EventStreamRPCConnection eventStreamRPCConnection =
                     IPCUtils.getEventStreamRpcConnection()) {
            GreengrassCoreIPCClient ipcClient =
                    new GreengrassCoreIPCClient(eventStreamRPCConnection);
            UpdateThingShadowResponseHandler responseHandler =
                    UpdateThingShadow.updateThingShadow(ipcClient, thingName,
 shadowName,
                            shadowPayload);
            CompletableFuture<UpdateThingShadowResponse> futureResponse =
                    responseHandler.getResponse();
            try {
                futureResponse.get(TIMEOUT_SECONDS, TimeUnit.SECONDS);
                System.out.printf("Successfully updated shadow: %s/%s%n", thingName,
 shadowName);
            } catch (TimeoutException e) {
                System.err.printf("Timeout occurred while updating shadow: %s/%s%n",
 thingName,
                        shadowName);
            } catch (ExecutionException e) {
                if (e.getCause() instanceof UnauthorizedError) {
                    System.err.printf("Unauthorized error while updating shadow: %s/
%s%n",
                            thingName, shadowName);
                } else {
                    throw e;
                }
            }
        } catch (InterruptedException e) {
            System.out.println("IPC interrupted.");
        } catch (ExecutionException e) {
```

```
            System.err.println("Exception occurred when using IPC.");
            e.printStackTrace();
            System.exit(1);
        }
    }

    public static UpdateThingShadowResponseHandler
 updateThingShadow(GreengrassCoreIPCClient greengrassCoreIPCClient, String
 thingName, String shadowName, byte[] shadowPayload) {
        UpdateThingShadowRequest updateThingShadowRequest = new
 UpdateThingShadowRequest();
        updateThingShadowRequest.setThingName(thingName);
        updateThingShadowRequest.setShadowName(shadowName);
        updateThingShadowRequest.setPayload(shadowPayload);
        return greengrassCoreIPCClient.updateThingShadow(updateThingShadowRequest,
                Optional.empty());
    }
}
```

Python (IPC client V1)

Example: Update a thing shadow

```
import awsiot.greengrasscoreipc
import awsiot.greengrasscoreipc.client as client
from awsiot.greengrasscoreipc.model import UpdateThingShadowRequest

TIMEOUT = 10

def sample_update_thing_shadow_request(thingName, shadowName, payload):
    try:
        # set up IPC client to connect to the IPC server
        ipc_client = awsiot.greengrasscoreipc.connect()

        # create the UpdateThingShadow request
        update_thing_shadow_request = UpdateThingShadowRequest()
        update_thing_shadow_request.thing_name = thingName
        update_thing_shadow_request.shadow_name = shadowName
        update_thing_shadow_request.payload = payload

        # retrieve the UpdateThingShadow response after sending the request to the
 IPC server
        op = ipc_client.new_update_thing_shadow()
        op.activate(update_thing_shadow_request)
```

```
        fut = op.get_response()

        result = fut.result(TIMEOUT)
        return result.payload

    except InvalidArgumentsError as e:
        # add error handling
    ...
    # except ConflictError | UnauthorizedError | ServiceError
```

JavaScript

Example: Update a thing shadow

```
import {
    UpdateThingShadowRequest
} from 'aws-iot-device-sdk-v2/dist/greengrasscoreipc/model';
import * as greengrasscoreipc from 'aws-iot-device-sdk-v2/dist/greengrasscoreipc';

class UpdateThingShadow {
    private ipcClient: greengrasscoreipc.Client;
    private thingName: string;
    private shadowName: string;
    private shadowDocumentStr: string;

    constructor() {
        // Define args parameters here

        this.thingName = "<define_your_own_thingName>";
        this.shadowName = "<define_your_own_shadowName>";
        this.shadowDocumentStr = "<define_your_own_payload>";

        this.bootstrap();
    }

    async bootstrap() {
        try {
            this.ipcClient = await getIpcClient();
        } catch (err) {
            // parse the error depending on your use cases
            throw err
        }

        try {
```

```
            await this.handleUpdateThingShadowOperation(
                this.thingName,
                this.shadowName,
                this.shadowDocumentStr);
        } catch (err) {
            // parse the error depending on your use cases
            throw err
        }
    }

    async handleUpdateThingShadowOperation(
        thingName: string,
        shadowName: string,
        payloadStr: string
    ) {
        const request: UpdateThingShadowRequest = {
            thingName: thingName,
            shadowName: shadowName,
            payload: payloadStr
        }
        // make the UpdateThingShadow request
        const response = await this.ipcClient.updateThingShadow(request);
    }
}

export async function getIpcClient() {
    try {
        const ipcClient = greengrasscoreipc.createClient();
        await ipcClient.connect()
            .catch(error => {
                // parse the error depending on your use cases
                throw error;
            });
        return ipcClient
    } catch (err) {
        // parse the error depending on your use cases
        throw err
    }
}

const startScript = new UpdateThingShadow();
```

DeleteThingShadow

Deletes the shadow for the specified thing.

Beginning in shadow manager v2.0.4, deleting a shadow increments the version number. For example, when you delete the shadow `MyThingShadow` at version 1, the version of the deleted shadow is 2. If you then recreate a shadow with the name `MyThingShadow`, the version for that shadow is 3.

Request

This operation's request has the following parameters:

`thingName` (Python: `thing_name`)

The name of the thing.

Type: `string`

`shadowName` (Python: `shadow_name`)

The name of the shadow. To specify the thing's classic shadow, set this parameter to an empty string ("").

> **Warning**
>
> The AWS IoT Greengrass service uses the `AWSManagedGreengrassV2Deployment` named shadow to manage deployments that target individual core devices. This named shadow is reserved for use by the AWS IoT Greengrass service. Do not update or delete this named shadow.

Type: `string`

Response

This operation's response has the following information:

`payload`

An empty response state document.

Errors

This operation can return the following errors.

`InvalidArgumentsError`

The local shadow service is unable to validate the request parameters. This can occur if the request contains malformed JSON or unsupported characters.

`ResourceNotFoundError`

The requested local shadow document can't be found.

`ServiceError`

An internal service error occurred, or the number of requests to the IPC service exceeded the limits specified in the `maxLocalRequestsPerSecondPerThing` and `maxTotalLocalRequestsRate` configuration parameters in the shadow manager component.

`UnauthorizedError`

The component's authorization policy doesn't include required permissions for this operation.

Examples

The following examples demonstrate how to call this operation in custom component code.

Java (IPC client V1)

Example: Delete a thing shadow

> **Note**
>
> This example uses an `IPCUtils` class to create a connection to the AWS IoT Greengrass Core IPC service. For more information, see Connect to the AWS IoT Greengrass Core IPC service.

```
package com.aws.greengrass.docs.samples.ipc;

import com.aws.greengrass.docs.samples.ipc.util.IPCUtils;
import software.amazon.awssdk.aws.greengrass.DeleteThingShadowResponseHandler;
import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClient;
```

```
import software.amazon.awssdk.aws.greengrass.model.DeleteThingShadowRequest;
import software.amazon.awssdk.aws.greengrass.model.DeleteThingShadowResponse;
import software.amazon.awssdk.aws.greengrass.model.ResourceNotFoundError;
import software.amazon.awssdk.aws.greengrass.model.UnauthorizedError;
import software.amazon.awssdk.eventstreamrpc.EventStreamRPCConnection;

import java.util.Optional;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public class DeleteThingShadow {

    public static final int TIMEOUT_SECONDS = 10;

    public static void main(String[] args) {
        // Use the current core device's name if thing name isn't set.
        String thingName = args[0].isEmpty() ? System.getenv("AWS_IOT_THING_NAME") :
 args[0];
        String shadowName = args[1];
        try (EventStreamRPCConnection eventStreamRPCConnection =
                     IPCUtils.getEventStreamRpcConnection()) {
            GreengrassCoreIPCClient ipcClient =
                    new GreengrassCoreIPCClient(eventStreamRPCConnection);
            DeleteThingShadowResponseHandler responseHandler =
                    DeleteThingShadow.deleteThingShadow(ipcClient, thingName,
 shadowName);
            CompletableFuture<DeleteThingShadowResponse> futureResponse =
                    responseHandler.getResponse();
            try {
                futureResponse.get(TIMEOUT_SECONDS, TimeUnit.SECONDS);
                System.out.printf("Successfully deleted shadow: %s/%s%n", thingName,
 shadowName);
            } catch (TimeoutException e) {
                System.err.printf("Timeout occurred while deleting shadow: %s/%s%n",
 thingName,
                        shadowName);
            } catch (ExecutionException e) {
                if (e.getCause() instanceof UnauthorizedError) {
                    System.err.printf("Unauthorized error while deleting shadow: %s/
%s%n",
                            thingName, shadowName);
                } else if (e.getCause() instanceof ResourceNotFoundError) {
```

```
                    System.err.printf("Unable to find shadow to delete: %s/%s%n",
 thingName,
                            shadowName);
                } else {
                    throw e;
                }
            }
        } catch (InterruptedException e) {
            System.out.println("IPC interrupted.");
        } catch (ExecutionException e) {
            System.err.println("Exception occurred when using IPC.");
            e.printStackTrace();
            System.exit(1);
        }
    }

    public static DeleteThingShadowResponseHandler
 deleteThingShadow(GreengrassCoreIPCClient greengrassCoreIPCClient, String
 thingName, String shadowName) {
        DeleteThingShadowRequest deleteThingShadowRequest = new
 DeleteThingShadowRequest();
        deleteThingShadowRequest.setThingName(thingName);
        deleteThingShadowRequest.setShadowName(shadowName);
        return greengrassCoreIPCClient.deleteThingShadow(deleteThingShadowRequest,
                Optional.empty());
    }
}
```

Python (IPC client V1)

Example: Delete a thing shadow

```
import awsiot.greengrasscoreipc
import awsiot.greengrasscoreipc.client as client
from awsiot.greengrasscoreipc.model import DeleteThingShadowRequest

TIMEOUT = 10

def sample_delete_thing_shadow_request(thingName, shadowName):
    try:
        # set up IPC client to connect to the IPC server
        ipc_client = awsiot.greengrasscoreipc.connect()

        # create the DeleteThingShadow request
```

```
        delete_thing_shadow_request = DeleteThingShadowRequest()
        delete_thing_shadow_request.thing_name = thingName
        delete_thing_shadow_request.shadow_name = shadowName

        # retrieve the DeleteThingShadow response after sending the request to the
 IPC server
        op = ipc_client.new_delete_thing_shadow()
        op.activate(delete_thing_shadow_request)
        fut = op.get_response()

        result = fut.result(TIMEOUT)
        return result.payload

    except InvalidArgumentsError as e:
        # add error handling
    ...
    # except ResourceNotFoundError | UnauthorizedError | ServiceError
```

JavaScript

Example: Delete a thing shadow

```
import {
    DeleteThingShadowRequest
} from 'aws-iot-device-sdk-v2/dist/greengrasscoreipc/model';
import * as greengrasscoreipc from 'aws-iot-device-sdk-v2/dist/greengrasscoreipc';

class DeleteThingShadow {
    private ipcClient: greengrasscoreipc.Client;
    private thingName: string;
    private shadowName: string;

    constructor() {
        // Define args parameters here
        this.thingName = "<define_your_own_thingName>";
        this.shadowName = "<define_your_own_shadowName>";
        this.bootstrap();
    }

    async bootstrap() {
        try {
            this.ipcClient = await getIpcClient();
        } catch (err) {
            // parse the error depending on your use cases
```

```
            throw err
        }

        try {
            await this.handleDeleteThingShadowOperation(this.thingName,
 this.shadowName)
        } catch (err) {
            // parse the error depending on your use cases
            throw err
        }
    }

    async handleDeleteThingShadowOperation(thingName: string, shadowName: string) {
        const request: DeleteThingShadowRequest = {
            thingName: thingName,
            shadowName: shadowName
        }
        // make the DeleteThingShadow request
        const response = await this.ipcClient.deleteThingShadow(request);
    }
}

export async function getIpcClient() {
    try {
        const ipcClient = greengrasscoreipc.createClient();
        await ipcClient.connect()
            .catch(error => {
                // parse the error depending on your use cases
                throw error;
            });
        return ipcClient
    } catch (err) {
        // parse the error depending on your use cases
        throw err
    }
}

const startScript = new DeleteThingShadow();
```

ListNamedShadowsForThing

List the named shadows for the specified thing.

Request

This operation's request has the following parameters:

`thingName` (Python: `thing_name`)

The name of the thing.

Type: `string`

`pageSize` (Python: `page_size`)

(Optional) The number of shadow names to return in each call.

Type: `integer`

Default: 25

Maximum: 100

`nextToken` (Python: `next_token`)

(Optional) The token to retrieve the next set of results. This value is returned on paged results and is used in the call that returns the next page.

Type: `string`

Response

This operation's response has the following information:

`results`

The list of shadow names.

Type: `array`

`timestamp`

(Optional) The date and time that the response was generated.

Type: `integer`

`nextToken` (Python: `next_token`)

(Optional) The token value to use in paged requests to retrieve the next page in the sequence. This token isn't present when there are no more shadow names to return.

Type: `string`

> **Note**
>
> If the requested page size exactly matches the number of shadow names in the response, then this token is present; however, when used, it returns an empty list.

Errors

This operation can return the following errors.

`InvalidArgumentsError`

The local shadow service is unable to validate the request parameters. This can occur if the request contains malformed JSON or unsupported characters.

`ResourceNotFoundError`

The requested local shadow document can't be found.

`ServiceError`

An internal service error occurred, or the number of requests to the IPC service exceeded the limits specified in the `maxLocalRequestsPerSecondPerThing` and `maxTotalLocalRequestsRate` configuration parameters in the shadow manager component.

`UnauthorizedError`

The component's authorization policy doesn't include required permissions for this operation.

Examples

The following examples demonstrate how to call this operation in custom component code.

Java (IPC client V1)

Example: List a thing's named shadows

> **Note**
>
> This example uses an `IPCUtils` class to create a connection to the AWS IoT Greengrass Core IPC service. For more information, see Connect to the AWS IoT Greengrass Core IPC service.

```
package com.aws.greengrass.docs.samples.ipc;

import com.aws.greengrass.docs.samples.ipc.util.IPCUtils;
import software.amazon.awssdk.aws.greengrass.GreengrassCoreIPCClient;
import
 software.amazon.awssdk.aws.greengrass.ListNamedShadowsForThingResponseHandler;
import software.amazon.awssdk.aws.greengrass.model.ListNamedShadowsForThingRequest;
import
 software.amazon.awssdk.aws.greengrass.model.ListNamedShadowsForThingResponse;
import software.amazon.awssdk.aws.greengrass.model.ResourceNotFoundError;
import software.amazon.awssdk.aws.greengrass.model.UnauthorizedError;
import software.amazon.awssdk.eventstreamrpc.EventStreamRPCConnection;

import java.util.ArrayList;
import java.util.List;
import java.util.Optional;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public class ListNamedShadowsForThing {

    public static final int TIMEOUT_SECONDS = 10;

    public static void main(String[] args) {
        // Use the current core device's name if thing name isn't set.
        String thingName = args[0].isEmpty() ? System.getenv("AWS_IOT_THING_NAME") :
 args[0];
        try (EventStreamRPCConnection eventStreamRPCConnection =
                     IPCUtils.getEventStreamRpcConnection()) {
```

```
            GreengrassCoreIPCClient ipcClient =
                    new GreengrassCoreIPCClient(eventStreamRPCConnection);
            List<String> namedShadows = new ArrayList<>();
            String nextToken = null;
            try {
                // Send additional requests until there's no pagination token in the
 response.
                do {
                    ListNamedShadowsForThingResponseHandler responseHandler =

 ListNamedShadowsForThing.listNamedShadowsForThing(ipcClient, thingName,
                                    nextToken, 25);
                    CompletableFuture<ListNamedShadowsForThingResponse>
 futureResponse =
                            responseHandler.getResponse();
                    ListNamedShadowsForThingResponse response =
                            futureResponse.get(TIMEOUT_SECONDS, TimeUnit.SECONDS);
                    List<String> responseNamedShadows = response.getResults();
                    namedShadows.addAll(responseNamedShadows);
                    nextToken = response.getNextToken();
                } while (nextToken != null);
                System.out.printf("Successfully got named shadows for thing %s: %s
%n", thingName,
                        String.join(",", namedShadows));
            } catch (TimeoutException e) {
                System.err.println("Timeout occurred while listing named shadows for
 thing: " + thingName);
            } catch (ExecutionException e) {
                if (e.getCause() instanceof UnauthorizedError) {
                    System.err.println("Unauthorized error while listing named
 shadows for " +
                            "thing: " + thingName);
                } else if (e.getCause() instanceof ResourceNotFoundError) {
                    System.err.println("Unable to find thing to list named shadows:
 " + thingName);
                } else {
                    throw e;
                }
            }
        } catch (InterruptedException e) {
            System.out.println("IPC interrupted.");
        } catch (ExecutionException e) {
            System.err.println("Exception occurred when using IPC.");
            e.printStackTrace();
```

```
            System.exit(1);
        }
    }

    public static ListNamedShadowsForThingResponseHandler
 listNamedShadowsForThing(GreengrassCoreIPCClient greengrassCoreIPCClient, String
 thingName, String nextToken, int pageSize) {
        ListNamedShadowsForThingRequest listNamedShadowsForThingRequest =
                new ListNamedShadowsForThingRequest();
        listNamedShadowsForThingRequest.setThingName(thingName);
        listNamedShadowsForThingRequest.setNextToken(nextToken);
        listNamedShadowsForThingRequest.setPageSize(pageSize);
        return
 greengrassCoreIPCClient.listNamedShadowsForThing(listNamedShadowsForThingRequest,
                Optional.empty());
    }
}
```

Python (IPC client V1)

Example: List a thing's named shadows

```
import awsiot.greengrasscoreipc
import awsiot.greengrasscoreipc.client as client
from awsiot.greengrasscoreipc.model import ListNamedShadowsForThingRequest

TIMEOUT = 10

def sample_list_named_shadows_for_thing_request(thingName, nextToken, pageSize):
    try:
        # set up IPC client to connect to the IPC server
        ipc_client = awsiot.greengrasscoreipc.connect()

        # create the ListNamedShadowsForThingRequest request
        list_named_shadows_for_thing_request = ListNamedShadowsForThingRequest()
        list_named_shadows_for_thing_request.thing_name = thingName
        list_named_shadows_for_thing_request.next_token = nextToken
        list_named_shadows_for_thing_request.page_size = pageSize

        # retrieve the ListNamedShadowsForThingRequest response after sending the
 request to the IPC server
        op = ipc_client.new_list_named_shadows_for_thing()
        op.activate(list_named_shadows_for_thing_request)
        fut = op.get_response()
```

```
        list_result = fut.result(TIMEOUT)

        # additional returned fields
        timestamp = list_result.timestamp
        next_token = result.next_token
        named_shadow_list = list_result.results

        return named_shadow_list, next_token, timestamp

    except InvalidArgumentsError as e:
        # add error handling
    ...
    # except ResourceNotFoundError | UnauthorizedError | ServiceError
```

JavaScript

Example: List a thing's named shadows

```
import {
    ListNamedShadowsForThingRequest
} from 'aws-iot-device-sdk-v2/dist/greengrasscoreipc/model';
import * as greengrasscoreipc from 'aws-iot-device-sdk-v2/dist/greengrasscoreipc';

class listNamedShadowsForThing {
    private ipcClient: greengrasscoreipc.Client;
    private thingName: string;
    private pageSizeStr: string;
    private nextToken: string;

    constructor() {
        // Define args parameters here
        this.thingName = "<define_your_own_thingName>";
        this.pageSizeStr = "<define_your_own_pageSize>";
        this.nextToken = "<define_your_own_token>";
        this.bootstrap();
    }

    async bootstrap() {
        try {
            this.ipcClient = await getIpcClient();
        } catch (err) {
            // parse the error depending on your use cases
            throw err
```

```
        }

        try {
            await this.handleListNamedShadowsForThingOperation(this.thingName,
                this.nextToken, this.pageSizeStr);
        } catch (err) {
            // parse the error depending on your use cases
            throw err
        }
    }

    async handleListNamedShadowsForThingOperation(
        thingName: string,
        nextToken: string,
        pageSizeStr: string
    ) {
        let request: ListNamedShadowsForThingRequest = {
            thingName: thingName,
            nextToken: nextToken,
        };
        if (pageSizeStr) {
            request.pageSize = parseInt(pageSizeStr);
        }
        // make the ListNamedShadowsForThing request
        const response = await this.ipcClient.listNamedShadowsForThing(request);
        const shadowNames = response.results;
    }
}

export async function getIpcClient(){
    try {
        const ipcClient = greengrasscoreipc.createClient();
        await ipcClient.connect()
            .catch(error => {
                // parse the error depending on your use cases
                throw error;
            });
        return ipcClient
    } catch (err) {
        // parse the error depending on your use cases
        throw err
    }
}
```

```
const startScript = new listNamedShadowsForThing();
```

Manage local deployments and components

> **Note**
>
> This feature is available for v2.6.0 and later of the Greengrass nucleus component.

Use the Greengrass CLI IPC service to manage local deployments and Greengrass components on the core device.

To use these IPC operations, include version 2.6.0 or later of the Greengrass CLI component as a dependency in your custom component. You can then use IPC operations in your custom components to do the following:

- Create local deployments to modify and configure Greengrass components on the core device.
- Restart and stop Greengrass components on the core device.
- Generate a password that you can use to sign in to the local debug console.

Topics

- Minimum SDK versions
- Authorization
- CreateLocalDeployment
- ListLocalDeployments
- GetLocalDeploymentStatus
- ListComponents
- GetComponentDetails
- RestartComponent
- StopComponent
- CreateDebugPassword

Minimum SDK versions

The following table lists the minimum versions of the SDKs that you can use to interact with the Greengrass CLI IPC service.

SDK	Minimum version
AWS IoT Greengrass Component SDK (C, C++, Rust)	v1.0.0
AWS IoT Device SDK for Java v2	v1.2.10
AWS IoT Device SDK for Python v2	v1.5.3
AWS IoT Device SDK for C++ v2	v1.17.0
AWS IoT Device SDK for JavaScript v2	v1.12.0

Authorization

To use the Greengrass CLI IPC service in a custom component, you must define authorization policies that allow your component to manage local deployments and components. For information about defining authorization policies, see Authorize components to perform IPC operations.

Authorization policies for the Greengrass CLI have the following properties.

IPC service identifier: `aws.greengrass.Cli`

Operation	Description	Resources
`aws.greengrass#CreateLocalDeployment`	Allows a component to create a local deployment on the core device.	*
`aws.greengrass#ListLocalDeployments`	Allows a component to list local deployments on the core device.	*
`aws.greengrass#GetLocalDeploymentStatus`	Allows a component to get the status of a local deployment on the core device.	A local deployment ID, or * to allow access to all local deployments.
`aws.greengrass#ListComponents`	Allows a component to list components on the core device.	*
`aws.greengrass#GetComponentDetails`	Allows a component to get details about a component on the core device.	A component name, such as `com.example.HelloWorld` , or * to allow access to all components.
`aws.greengrass#RestartComponent`	Allows a component to restart a component on the core device.	A component name, such as `com.example.HelloWorld` , or * to allow access to all components.
`aws.greengrass#StopComponent`	Allows a component to stop a component on the core device.	A component name, such as `com.example.HelloWorld` , or * to allow access to all components.
`aws.greengrass#CreateDebugPassword`	Allows a component to generate a password to use to sign in to the local debug console component.	*

Example authorization policy

The following example authorization policies allow a component to create local deployments, view all local deployments and components, and restart and stop a component named `com.example.HelloWorld`.

```
{
  "accessControl": {
    "aws.greengrass.Cli": {
      "com.example.MyLocalManagerComponent:cli:1": {
        "policyDescription": "Allows access to create local deployments and view
 deployments and components.",
        "operations": [
          "aws.greengrass#CreateLocalDeployment",
          "aws.greengrass#ListLocalDeployments",
          "aws.greengrass#GetLocalDeploymentStatus",
          "aws.greengrass#ListComponents",
          "aws.greengrass#GetComponentDetails"
        ],
        "resources": [
          "*"
        ]
      }
    },
    "aws.greengrass.Cli": {
      "com.example.MyLocalManagerComponent:cli:2": {
        "policyDescription": "Allows access to restart and stop the Hello World
 component.",
        "operations": [
          "aws.greengrass#RestartComponent",
          "aws.greengrass#StopComponent"
        ],
        "resources": [
          "com.example.HelloWorld"
        ]
      }
    }
  }
}
```

CreateLocalDeployment

Create or update a local deployment using specified component recipes, artifacts, and runtime arguments.

This operation provides the same functionality as the deployment create command in the Greengrass CLI.

Request

This operation's request has the following parameters:

`recipeDirectoryPath` (Python: `recipe_directory_path`)

(Optional) The absolute path to the folder that contains component recipe files.

`artifactsDirectoryPath` (Python: `artifacts_directory_path`)

(Optional) The absolute path to the folder that contains the artifact files to include in the deployment. The artifacts folder must contain the following folder structure:

```
/path/to/artifact/folder/component-name/component-version/artifacts
```

`rootComponentVersionsToAdd` (Python: `root_component_versions_to_add`)

(Optional) The component versions to install on the core device. This object, `ComponentToVersionMap`, is a map that contains the following key-value pairs:

`key`

The name of the component.

`value`

The version of the component.

`rootComponentsToRemove` (Python: `root_components_to_remove`)

(Optional) The components to uninstall from the core device. Specify a list where each entry is the name of a component.

`componentToConfiguration` (Python: `component_to_configuration`)

(Optional) The configuration updates for each component in the deployment. This object, `ComponentToConfiguration`, is a map that contains the following key-value pairs:

`key`

The name of the component.

`value`

The configuration update JSON object for the component. The JSON object must have the following format.

```
{
  "MERGE": {
    "config-key": "config-value"
  },
  "RESET": [
    "path/to/reset/"
  ]
}
```

For more information about configuration updates, see Update component configurations.

`componentToRunWithInfo` (Python: `component_to_run_with_info`)

(Optional) The runtime configuration for each component in the deployment. This configuration includes the system user that owns each component's processes and the system limits to apply to each component. This object, `ComponentToRunWithInfo`, is a map that contains the following key-value pairs:

`key`

The name of the component.

`value`

The runtime configuration for the component. If you omit a runtime configuration parameter, the AWS IoT Greengrass Core software uses the default values that you configure on the Greengrass nucleus. This object, `RunWithInfo`, contains the following information:

`posixUser` (Python: `posix_user`)

(Optional) The POSIX system user and, optionally, group to use to run this component on Linux core devices. The user, and group if specified, must exist on each Linux core device. Specify the user and group separated by a colon (`:`) in the following format: `user:group`. The group is optional. If you don't specify a group, the AWS IoT Greengrass Core software uses the primary group for the user. For more information, see Configure the user that runs components.

`windowsUser` (Python: `windows_user`)

(Optional) The Windows user to use to run this component on Windows core devices. The user must exist on each Windows core device, and its name and password must be stored in the LocalSystem account's Credentials Manager instance. For more information, see Configure the user that runs components.

`systemResourceLimits` (Python: `system_resource_limits`)

(Optional) The system resource limits to apply to this component's processes. You can apply system resource limits to generic and non-containerized Lambda components. For more information, see Configure system resource limits for components.

AWS IoT Greengrass doesn't currently support this feature on Windows core devices.

This object, `SystemResourceLimits`, contains the following information:

`cpus`

(Optional) The maximum amount of CPU time that this component's processes can use on the core device. A core device's total CPU time is equivalent to the device's number of CPU cores. For example, on a core device with 4 CPU cores, you can set this value to 2 to limit this component's processes to 50 percent usage of each CPU core. On a device with 1 CPU core, you can set this value to `0.25` to limit this component's processes to 25 percent usage of the CPU. If you set this value to a number greater than the number of CPU cores, the AWS IoT Greengrass Core software doesn't limit the component's CPU usage.

`memory`

(Optional) The maximum amount of RAM (in kilobytes) that this component's processes can use on the core device.

`groupName` (Python: `group_name`)

(Optional) The name of the thing group to target with this deployment.

`failureHandlingPolicy` (Python: `failure_handling_policy`)

(Optional) The failure handling policy that determines how the AWS IoT Greengrass Core software responds if the deployment fails.

`ROLLBACK`

The core device rolls back to its previous configuration when the deployment fails.

`DO_NOTHING`

The core device keeps the changes from the failed deployment.

> **Note**
>
> The AWS IoT Greengrass Core software ignores the failure handling policy for bootstrap deployments. A bootstrap deployment updates the Greengrass nucleus version or includes a component with a bootstrap lifecycle step. If a bootstrap deployment fails, the core device rolls back to its previous configuration. The rollback happens even if you set the failure handling policy to `DO_NOTHING`.

Response

This operation's response has the following information:

`deploymentId` (Python: `deployment_id`)

The ID of the local deployment that the request created.

ListLocalDeployments

Gets the status of the last 10 local deployments.

This operation provides the same functionality as the deployment list command in the Greengrass CLI.

Request

This operation's request doesn't have any parameters.

Response

This operation's response has the following information:

`localDeployments` (Python: `local_deployments`)

The list of local deployments. Each object in this list is a `LocalDeployment` object, which contains the following information:

`deploymentId` (Python: `deployment_id`)

The ID of the local deployment.

`status`

The status of the local deployment. This enum, `DeploymentStatus`, has the following values:

- `QUEUED`
- `IN_PROGRESS`
- `SUCCEEDED`
- `FAILED`

GetLocalDeploymentStatus

Gets the status of a local deployment.

This operation provides the same functionality as the deployment status command in the Greengrass CLI.

Request

This operation's request has the following parameters:

`deploymentId` (Python: `deployment_id`)

The ID of the local deployment to get.

Response

This operation's response has the following information:

`deployment`

The local deployment. This object, `LocalDeployment`, contains the following information:

`deploymentId` (Python: `deployment_id`)

The ID of the local deployment.

`status`

The status of the local deployment. This enum, `DeploymentStatus`, has the following values:

- `QUEUED`
- `IN_PROGRESS`
- `SUCCEEDED`
- `FAILED`

ListComponents

Gets the name, version, status, and configuration of each root component on the core device. A *root component* is a component that you specify in a deployment. This response doesn't include components that are installed as dependencies of other components.

This operation provides the same functionality as the component list command in the Greengrass CLI.

Request

This operation's request doesn't have any parameters.

Response

This operation's response has the following information:

`components`

The list of root components on the core device. Each object in this list is a `ComponentDetails` object, which contains the following information:

`componentName` (Python: `component_name`)

The name of the component.

`version`

The version of the component.

`state`

The state of the component. This state can be one of the following:

- BROKEN
- ERRORED
- FINISHED
- INSTALLED
- NEW
- RUNNING
- STARTING
- STOPPING

`configuration`

The component's configuration as a JSON object.

GetComponentDetails

Gets the version, status, and configuration of a component on the core device.

This operation provides the same functionality as the component details command in the Greengrass CLI.

Request

This operation's request has the following parameters:

`componentName` (Python: `component_name`)

The name of the component to get.

Response

This operation's response has the following information:

`componentDetails` (Python: `component_details`)

The component's details. This object, `ComponentDetails`, contains the following information:

`componentName` (Python: `component_name`)

The name of the component.

`version`

The version of the component.

`state`

The state of the component. This state can be one of the following:

- `BROKEN`
- `ERRORED`
- `FINISHED`
- `INSTALLED`
- `NEW`
- `RUNNING`
- `STARTING`
- `STOPPING`

`configuration`

The component's configuration as a JSON object.

RestartComponent

Restarts a component on the core device.

> **Note**
>
> While you can restart any component, we recommend that you restart only generic components.

This operation provides the same functionality as the component restart command in the Greengrass CLI.

Request

This operation's request has the following parameters:

`componentName` (Python: `component_name`)

The name of the component.

Response

This operation's response has the following information:

`restartStatus` (Python: `restart_status`)

The status of the restart request. The request status can be one of the following:

- `SUCCEEDED`
- `FAILED`

`message`

A message about why the component failed to restart, if the request failed.

Examples

The following examples demonstrate how to call this operation in custom component code.

Rust

Example: Restart a component

```
use gg_sdk::Sdk;

fn main() {
    let sdk = Sdk::init();
    sdk.connect().expect("Failed to establish IPC connection");

    let component_name = "com.example.HelloWorld";

    sdk.restart_component(component_name)
        .expect("Failed to restart component");

    println!("Successfully requested restart for component: {component_name}");
}
```

C

Example: Restart a component

```
#include <gg/error.h>
#include <gg/ipc/client.h>
#include <gg/sdk.h>
```

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    gg_sdk_init();

    GgError err = ggipc_connect();
    if (err != GG_ERR_OK) {
        fprintf(stderr, "Failed to establish IPC connection.\n");
        exit(-1);
    }

    GgBuffer component_name = GG_STR("com.example.HelloWorld");

    err = ggipc_restart_component(component_name);
    if (err != GG_ERR_OK) {
        fprintf(
            stderr,
            "Failed to restart component: %.*s\n",
            (int) component_name.len,
            component_name.data
        );
        exit(-1);
    }

    printf(
        "Successfully requested restart for component: %.*s\n",
        (int) component_name.len,
        component_name.data
    );
}
```

C++ (Component SDK)

Example: Restart a component

```
#include <gg/ipc/client.hpp>
#include <iostream>

int main() {
    auto &client = gg::ipc::Client::get();

    auto error = client.connect();
    if (error) {
```

```
        std::cerr << "Failed to establish IPC connection.\n";
        exit(-1);
    }

    std::string_view component_name = "com.example.HelloWorld";

    error = client.restart_component(component_name);
    if (error) {
        std::cerr << "Failed to restart component: " << component_name << "\n";
        exit(-1);
    }

    std::cout << "Successfully requested restart for component: "
              << component_name << "\n";
}
```

StopComponent

Stops a component's processes on the core device.

> **Note**
>
> While you can stop any component, we recommend that you stop only generic components.

This operation provides the same functionality as the component stop command in the Greengrass CLI.

Request

This operation's request has the following parameters:

`componentName` (Python: `component_name`)

The name of the component.

Response

This operation's response has the following information:

`stopStatus` (Python: `stop_status`)

The status of the stop request. The request status can be one of the following:

- `SUCCEEDED`
- `FAILED`

`message`

A message about why the component failed to stop, if the request failed.

CreateDebugPassword

Generates a random password that you can use to sign in to the local debug console component. The password expires 8 hours after it is generated.

This operation provides the same functionality as the get-debug-password command in the Greengrass CLI.

Request

This operation's request doesn't have any parameters.

Response

This operation's response has the following information:

`username`

The user name to use to sign in.

`password`

The password to use to sign in.

`passwordExpiration` (Python: `password_expiration`)

The time when the password expires.

`certificateSHA256Hash` (Python: `certificate_sha256_hash`)

The SHA-256 fingerprint for the self-signed certificate that the local debug console uses when HTTPS is enabled. When you open the local debug console, use this fingerprint to verify that the certificate is legitimate and the connection is secure.

`certificateSHA1Hash` (Python: `certificate_sha1_hash`)

The SHA-1 fingerprint for the self-signed certificate that the local debug console uses when HTTPS is enabled. When you open the local debug console, use this fingerprint to verify that the certificate is legitimate and the connection is secure.

Authenticate and authorize client devices

> **Note**
>
> This feature is available for v2.6.0 and later of the Greengrass nucleus component.

Use the client device auth IPC service to develop a custom local broker component where local IoT devices, such as client devices, can connect.

To use these IPC operations, include version 2.2.0 or later of the client device auth component as a dependency in your custom component. You can then use IPC operations in your custom components to do the following:

- Verify the identity of client devices that connect to the core device.
- Create a session for a client device to connect to the core device.
- Verify whether a client device has permission to perform an action.
- Receive a notification when the core device's server certificate rotates.

Topics

- Minimum SDK versions
- Authorization
- VerifyClientDeviceIdentity
- GetClientDeviceAuthToken
- AuthorizeClientDeviceAction
- SubscribeToCertificateUpdates

Minimum SDK versions

The following table lists the minimum versions of the AWS IoT Device SDK that you must use to interact with the client device auth IPC service.

SDK	Minimum version	
AWS IoT Device SDK for Java v2	v1.9.3	
AWS IoT Device SDK for Python v2	v1.11.3	
AWS IoT Device SDK for C++ v2	v1.18.3	
AWS IoT Device SDK for JavaScript v2	v1.12.0	

Authorization

To use the client device auth IPC service in a custom component, you must define authorization policies that allow your component to perform these operations. For information about defining authorization policies, see Authorize components to perform IPC operations.

Authorization policies for client device authentication and authorization have the following properties.

IPC service identifier: `aws.greengrass.clientdevices.Auth`

Operation	Description	Resources
`aws.greengrass#VerifyClientDeviceIdentity`	Allows a component to verify the identity of a client device.	*

Operation	Description	Resources
aws.greengrass#GetClientDeviceAuthToken	Allows a component to validate a client device's credentials and create a session for that client device.	*
aws.greengrass#AuthorizeClientDeviceAction	Allows a component to verify whether a client device has permission to perform an action.	*
aws.greengrass#SubscribeToCertificateUpdates	Allows a component to receive notifications when the core device's server certificate rotates.	*
*	Allows a component to perform all client device auth IPC service operations.	*

Authorization policy examples

You can reference the following authorization policy example to help you configure authorization policies for your components.

Example authorization policy

The following example authorization policy allows a component to perform all client device auth IPC operations.

```
{
  "accessControl": {
    "aws.greengrass.clientdevices.Auth": {
      "com.example.MyLocalBrokerComponent:clientdevices:1": {
        "policyDescription": "Allows access to authenticate and authorize client
 devices.",
        "operations": [
          "aws.greengrass#VerifyClientDeviceIdentity",
```

```
          "aws.greengrass#GetClientDeviceAuthToken",
          "aws.greengrass#AuthorizeClientDeviceAction",
          "aws.greengrass#SubscribeToCertificateUpdates"
        ],
        "resources": [
          "*"
        ]
      }
    }
  }
}
```

VerifyClientDeviceIdentity

Verify the identity of a client device. This operation verifies whether the client device is a valid AWS IoT thing.

Request

This operation's request has the following parameters:

`credential`

The client device's credentials. This object, `ClientDeviceCredential`, contains the following information:

`clientDeviceCertificate` (Python: `client_device_certificate`)

The client device's X.509 device certificate.

Response

This operation's response has the following information:

`isValidClientDevice` (Python: `is_valid_client_device`)

Whether the client device's identity is valid.

GetClientDeviceAuthToken

Validates a client device's credentials and creates a session for the client device. This operation returns a session token that you can use in subsequent requests to authorize client device actions.

To successfully connect a client device, the client device auth component must grant the `mqtt:connect` permission for the client ID that the client device uses.

Request

This operation's request has the following parameters:

`credential`

The client device's credentials. This object, `CredentialDocument`, contains the following information:

`mqttCredential` (Python: `mqtt_credential`)

The client device's MQTT credentials. Specify the client ID and certificate that the client device uses to connect. This object, `MQTTCredential`, contains the following information:

`clientId` (Python: `client_id`)

The client ID to use to connect.

`certificatePem` (Python: `certificate_pem`)

The X.509 device certificate to use to connect.

`username`

> **Note**
>
> This property isn't currently used.

`password`

> **Note**
>
> This property isn't currently used.

Response

This operation's response has the following information:

`clientDeviceAuthToken` (Python: `client_device_auth_token`)

The session token for the client device. You can use this session token in subsequent requests to authorize this client device's actions.

AuthorizeClientDeviceAction

Verify whether a client device has permission to perform an action on a resource. *Client device authorization policies* specify the permissions that client devices can perform while connected to a core device. You define client device authorization policies when you configure the client device auth component.

Request

This operation's request has the following parameters:

`clientDeviceAuthToken` (Python: `client_device_auth_token`)

The session token for the client device.

`operation`

The operation to authorize.

`resource`

The resource where the client device performs the operation.

Response

This operation's response has the following information:

`isAuthorized` (Python: `is_authorized`)

Whether the client device is authorized to perform the operation on the resource.

SubscribeToCertificateUpdates

Subscribe to receive the core device's new server certificate each time it rotates. When the server certificate rotates, brokers must reload using the new server certificate.

The client device auth component rotates server certificates every 7 days by default. You can configure the rotation interval to between 2 and 10 days.

This operation is a subscription operation where you subscribe to a stream of event messages. To use this operation, define a stream response handler with functions that handle event messages, errors, and stream closure. For more information, see Subscribe to IPC event streams.

Event message type: `CertificateUpdateEvent`

Request

This operation's request has the following parameters:

`certificateOptions` (Python: `certificate_options`)

The types of certificate updates to subscribe to. This object, `CertificateOptions`, contains the following information:

`certificateType` (Python: `certificate_type`)

The type of certificate updates to subscribe to. Choose the following option:

- `SERVER`

Response

This operation's response has the following information:

`messages`

The stream of messages. This object, `CertificateUpdateEvent`, contains the following information:

`certificateUpdate` (Python: `certificate_update`)

The information about the new certificate. This object, `CertificateUpdate`, contains the following information:

`certificate`

The certificate.

`privateKey` (Python: `private_key`)

The certificate's private key.

`publicKey` (Python: `public_key`)

The certificate's public key.

`caCertificates` (Python: `ca_certificates`)

The list of certificate authority (CA) certificates in the certificate's CA certificate chain.

Interact with local IoT devices

Client devices are local IoT devices that connect to and communicate with a Greengrass core device over MQTT. You can connect client devices to core devices to do the following:

- Interact with MQTT messages in Greengrass components.
- Relay messages and data between client devices and AWS IoT Core.
- Interact with client device shadows in Greengrass components.
- Sync client devices shadows with AWS IoT Core.
- Use IPv6 for local messaging.

To connect to a core device, client devices can use *cloud discovery*. Client devices connect to the AWS IoT Greengrass cloud service to retrieve information about core devices to which they can connect. Then, they can connect to a core device to process their messages and sync their data with the AWS IoT Core cloud service.

You can follow a tutorial that walks through how to configure a core device to connect and communicate with an AWS IoT thing. This tutorial also explores how to develop a custom Greengrass component that interacts with client devices. For more information, see Tutorial: Interact with local IoT devices over MQTT.

Topics

- AWS-provided client device components
- Connect client devices to core devices
- Relay MQTT messages between client devices and AWS IoT Core
- Interact with client devices in components
- Interact with and sync client device shadows
- Use IPv6 for local messaging
- Troubleshooting client devices

AWS-provided client device components

AWS IoT Greengrass provides the following public components that you can deploy to core devices. These components enable client devices to connect and communicate with a core device.

Note

Several AWS-provided components depend on specific minor versions of the Greengrass nucleus. Because of this dependency, you need to update these components when you update the Greengrass nucleus to a new minor version. For information about the specific versions of the nucleus that each component depends on, see the corresponding component topic. For more information about updating the nucleus, see Update the AWS IoT Greengrass Core software (OTA).

When a component has a component type of both generic and Lambda, the current version of the component is the generic type and a previous version of the component is the Lambda type.

Component	Description	Component type	Supported OS	Open source
Client device auth	Enables local IoT devices, called client devices, to connect to the core device.	Plugin	Linux, Windows	Yes
IP detector	Reports MQTT broker connectivity information to AWS IoT Greengras s, so client devices can discover how to connect.	Plugin	Linux, Windows	Yes
MQTT bridge	Relays MQTT messages between	Plugin	Linux, Windows	Yes

Component	Description	Component type	Supported OS	Open source
	client devices, local AWS IoT Greengras s publish/s ubscribe, and AWS IoT Core.			
MQTT 3.1.1 broker (Moquette)	Runs an MQTT 3.1.1 broker that handles messages between client devices and the core device.	Plugin	Linux, Windows	Yes
MQTT 5 broker (EMQX)	Runs an MQTT 5 broker that handles messages between client devices and the core device.	Generic	Linux, Windows	No

Component	Description	Component type	Supported OS	Open source
Shadow manager	Enables interaction with shadows on the core device. It manages shadow document storage and also the synchroni zation of local shadow states with the AWS IoT Device Shadow service.	Plugin	Linux, Windows	Yes

Connect client devices to core devices

You can configure *cloud discovery* to connect client devices to core devices. When you configure cloud discovery, client devices can connect to the AWS IoT Greengrass cloud service to retrieve information about core devices to which they can connect. Then, the client devices can attempt to connect to each core device until they successfully connect.

To use cloud discovery, you must do the following:

- Associate client devices to the core devices to which they can connect.
- Specify the MQTT broker endpoints where client devices can connect to each core device.
- Deploy components to the core device that enable support for client devices.

  You can also deploy optional components to do the following:

- Relay messages between client devices, Greengrass components, and the AWS IoT Core cloud service.
- Automatically manage core device MQTT broker endpoints for you.
- Manage local client device shadows and synchronize shadows with the AWS IoT Core cloud service.

You must also review and update the core device's AWS IoT policy to verify that it has the permissions required to connect client devices. For more information, see Requirements.

After you configure cloud discovery, you can test communications between a client device and a core device. For more information, see Test client device communications.

Topics

- Requirements
- Greengrass components for client device support
- Configure cloud discovery (console)
- Configure cloud discovery (AWS CLI)
- Associate client devices
- Authenticating clients while offline
- Manage core device endpoints
- Choose an MQTT broker
- Connecting client devices to an AWS IoT Greengrass Core device with an MQTT broker
- Test client device communications
- Greengrass discovery RESTful API

Requirements

To connect client devices to a core device, you must have the following:

- The core device must run Greengrass nucleus v2.2.0 or later.
- The Greengrass service role associated with AWS IoT Greengrass for your AWS account in the AWS Region where the core device operates. For more information, see Configure the Greengrass service role.
- The core device's AWS IoT policy must allow the following permissions:

- `greengrass:PutCertificateAuthorities`
- `greengrass:VerifyClientDeviceIdentity`
- `greengrass:VerifyClientDeviceIoTCertificateAssociation`
- `greengrass:GetConnectivityInfo`
- `greengrass:UpdateConnectivityInfo` – (Optional) This permission is required to use the IP detector component, which reports the core device's network connectivity information to the AWS IoT Greengrass cloud service.
- `iot:GetThingShadow`, `iot:UpdateThingShadow`, and `iot:DeleteThingShadow` – (Optional) These permissions are required to use the shadow manager component to sync client device shadows with AWS IoT Core. This feature requires Greengrass nucleus v2.6.0 or later, shadow manager v2.2.0 or later, and MQTT bridge v2.2.0 or later.

For more information, see Configure the AWS IoT thing policy.

> **Note**
>
> If you used the default AWS IoT policy when you installed the AWS IoT Greengrass Core software, the core device has an AWS IoT policy that allows access to all AWS IoT Greengrass actions (`greengrass:*`).

- AWS IoT things that you can connect as client devices. For more information, see Create AWS IoT resources in the *AWS IoT Core Developer Guide*.
- The client device must connect using a client ID. A client ID is a thing name. No other client ID will be accepted.
- Each client device's AWS IoT policy must allow the `greengrass:Discover` permission. For more information, see Minimal AWS IoT policy for client devices.

Topics

- Configure the Greengrass service role
- Configure the AWS IoT thing policy

Configure the Greengrass service role

The Greengrass service role is an AWS Identity and Access Management (IAM) service role that authorizes AWS IoT Greengrass to access resources from AWS services on your behalf. This role

makes it possible for AWS IoT Greengrass to verify the identity of client devices and manage core device connectivity information.

If you haven't previously set up the Greengrass service role in this Region, you must associate a Greengrass service role with AWS IoT Greengrass for your AWS account in this Region.

When you use the **Configure core device discovery** page in the AWS IoT Greengrass console, AWS IoT Greengrass sets up the Greengrass service role for you. Otherwise, you can manually set it up using the AWS IoT console or AWS IoT Greengrass API.

In this section, you check whether the Greengrass service role is set up. If it isn't set up, you create a new Greengrass service role to associate with AWS IoT Greengrass for your AWS account in this Region.

Configure the Greengrass service role (console)

1. Check if the Greengrass service role is associated with AWS IoT Greengrass for your AWS account in this Region. Do the following:

   a. Navigate to the AWS IoT console.

   b. In the navigation pane, choose **Settings**.

   c. In the **Greengrass service role** section, find **Current service role** to see whether a Greengrass service role is associated.

      If you have a Greengrass service role associated, you meet this requirement to use the IP detector component. Skip to Configure the AWS IoT thing policy.

2. If the Greengrass service role isn't associated with AWS IoT Greengrass for your AWS account in this Region, create a Greengrass service role and associate it. Do the following:

   a. Navigate to the IAM console.

   b. Choose **Roles**.

   c. Choose **Create role**.

   d. On the **Create role** page, do the following:

      i. Under **Trusted entity type**, choose **AWS service**.

      ii. Under **Use case**, **Use cases for other AWS services**, choose **Greengrass**, select **Greengrass**. This option specifies to add AWS IoT Greengrass as a trusted entity that can assume this role.

iii. Choose **Next**.

iv. Under **Permissions policies**, select the **AWSGreengrassResourceAccessRolePolicy** to attach to the role.

v. Choose **Next**.

vi. In **Role name**, enter a name for the role, such as **`Greengrass_ServiceRole`**.

vii. Choose **Create role**.

e. Navigate to the AWS IoT console.

f. In the navigation pane, choose **Settings**.

g. In the **Greengrass service role** section, choose **Attach role**.

h. In the **Update Greengrass service role** modal, select the IAM role that you created, and then choose **Attach role**.

Configure the Greengrass service role (AWS CLI)

1. Check if the Greengrass service role is associated with AWS IoT Greengrass for your AWS account in this Region.

```
aws greengrassv2 get-service-role-for-account
```

   If the Greengrass service role is associated, the operation returns a response that contains information about the role.

   If you have a Greengrass service role associated, you meet this requirement to use the IP detector component. Skip to Configure the AWS IoT thing policy.

2. If the Greengrass service role isn't associated with AWS IoT Greengrass for your AWS account in this Region, create a Greengrass service role and associate it. Do the following:

   a. Create a role with a trust policy that allows AWS IoT Greengrass to assume the role. This example creates a role named `Greengrass_ServiceRole`, but you can use a different name. We recommend that you also include the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in your trust policy to help prevent the *confused deputy* security problem. The condition context keys restrict access to allow only those requests that come from the specified account and Greengrass workspace. For more information about the confused deputy problem, see Cross-service confused deputy prevention.

Linux or Unix

```
aws iam create-role --role-name Greengrass_ServiceRole --assume-role-policy-
document '{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "greengrass.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "ArnLike": {
          "aws:SourceArn": "arn:aws:greengrass:region:account-id:*"
        },
        "StringEquals": {
          "aws:SourceAccount": "account-id"
        }
      }
    }
  ]
}'
```

Windows Command Prompt (CMD)

```
aws iam create-role --role-name Greengrass_ServiceRole --assume-role-policy-
document "{\\"Version\\":\\"2012-10-17       \\",\\"Statement\\":[{\\"Effect
\\":\\"Allow\\",\\"Principal\\":{\\"Service\\":\\"greengrass.amazonaws.com
\\"},\\"Action\\":\\"sts:AssumeRole\\",\\"Condition\\":{\\"ArnLike\\":
{\\"aws:SourceArn\\":\\"arn:aws:greengrass:region:account-id:*\\"},\
\\"StringEquals\\":{\\"aws:SourceAccount\\":\\"account-id\\"}}}]}"
```

PowerShell

```
aws iam create-role --role-name Greengrass_ServiceRole --assume-role-policy-
document '{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
```

```
      "Principal": {
        "Service": "greengrass.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "ArnLike": {
          "aws:SourceArn": "arn:aws:greengrass:region:account-id:*"
        },
        "StringEquals": {
          "aws:SourceAccount": "account-id"
        }
      }
    }
  ]
}'
```

b. Copy the role ARN from the role metadata in the output. You use the ARN to associate the role with your account.

c. Attach the `AWSGreengrassResourceAccessRolePolicy` policy to the role.

```
aws iam attach-role-policy --role-name Greengrass_ServiceRole --policy-arn
 arn:aws:iam::aws:policy/service-role/AWSGreengrassResourceAccessRolePolicy
```

d. Associate the Greengrass service role with AWS IoT Greengrass for your AWS account. Replace *role-arn* with the ARN of the service role.

```
aws greengrassv2 associate-service-role-to-account --role-arn role-arn
```

The operation returns the following response if it succeeds.

```
{
  "associatedAt": "timestamp"
}
```

Configure the AWS IoT thing policy

Core devices use X.509 device certificates to authorize connections to AWS. You attach AWS IoT policies to device certificates to define the permissions for a core device. For more information, see AWS IoT policies for data plane operations and Minimal AWS IoT policy to support client devices.

To connect client devices to a core device, the core device's AWS IoT policy must allow the following permissions:

- `greengrass:PutCertificateAuthorities`
- `greengrass:VerifyClientDeviceIdentity`
- `greengrass:VerifyClientDeviceIoTCertificateAssociation`
- `greengrass:GetConnectivityInfo`
- `greengrass:UpdateConnectivityInfo` – (Optional) This permission is required to use the IP detector component, which reports the core device's network connectivity information to the AWS IoT Greengrass cloud service.
- `iot:GetThingShadow`, `iot:UpdateThingShadow`, and `iot:DeleteThingShadow` – (Optional) These permissions are required to use the shadow manager component to sync client device shadows with AWS IoT Core. This feature requires Greengrass nucleus v2.6.0 or later, shadow manager v2.2.0 or later, and MQTT bridge v2.2.0 or later.

In this section, you review the AWS IoT policies for your core device and add any required permissions that are missing. If you used the AWS IoT Greengrass Core software installer to provision resources, your core device has an AWS IoT policy that allows access to all AWS IoT Greengrass actions (`greengrass:*`). In this case, you must update the AWS IoT policy only if you plan to deploy the shadow manager component to sync device shadows with AWS IoT Core. Otherwise, you can skip this section.

Configure the AWS IoT thing policy (console)

1. In the AWS IoT Greengrass console navigation menu, choose **Core devices**.
2. On the **Core devices** page, choose the core device to update.
3. On the core device details page, choose the link to the core device's **Thing**. This link opens the thing details page in the AWS IoT console.
4. On the thing details page, choose **Certificates**.
5. In the **Certificates** tab, choose the thing's active certificate.
6. On the certificate details page, choose **Policies**.
7. In the **Policies** tab, choose the AWS IoT policy to review and update. You can add the required permissions to any policy that is attached to the core device's active certificate.

> **Note**
>
> If you used the AWS IoT Greengrass Core software installer to provision resources, you have two AWS IoT policies. We recommend that you choose the policy named **GreengrassV2IoTThingPolicy**, if it exists. Core devices that you create with the quick installer use this policy name by default. If you add permissions to this policy, you are also granting these permissions to other core devices that use this policy.

8. In the policy overview, choose **Edit active version**.
9. Review the policy for the required permissions, and add any required permissions that are missing.

   - `greengrass:PutCertificateAuthorities`
   - `greengrass:VerifyClientDeviceIdentity`
   - `greengrass:VerifyClientDeviceIoTCertificateAssociation`
   - `greengrass:GetConnectivityInfo`
   - `greengrass:UpdateConnectivityInfo` – (Optional) This permission is required to use the IP detector component, which reports the core device's network connectivity information to the AWS IoT Greengrass cloud service.
   - `iot:GetThingShadow`, `iot:UpdateThingShadow`, and `iot:DeleteThingShadow` – (Optional) These permissions are required to use the shadow manager component to sync client device shadows with AWS IoT Core. This feature requires Greengrass nucleus v2.6.0 or later, shadow manager v2.2.0 or later, and MQTT bridge v2.2.0 or later.
10. (Optional) To allow the core device to sync shadows with AWS IoT Core, add the following statement to the policy. If you plan to interact with client device shadows, but not sync them with AWS IoT Core, skip this step. Replace *`region`* and *`account-id`* with the Region that you use and your AWS account number.

    - This example statement allows access to all things' device shadows. To follow best security practices, you can restrict access to only the core device and the client devices that you connect to the core device. For more information, see Minimal AWS IoT policy to support client devices.

```
{
  "Effect": "Allow",
```

```
  "Action": [
    "iot:GetThingShadow",
    "iot:UpdateThingShadow",
    "iot:DeleteThingShadow"
  ],
  "Resource": [
    "arn:aws:iot:region:account-id:thing/*"
  ]
}
```

After you add this statement, the policy document might look similar to the following example.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:Connect",
        "iot:Publish",
        "iot:Subscribe",
        "iot:Receive",
        "greengrass:*"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "iot:GetThingShadow",
        "iot:UpdateThingShadow",
        "iot:DeleteThingShadow"
      ],
      "Resource": [
        "arn:aws:iot:us-east-1:123456789012:thing/*"
      ]
    }
  ]
}
```

11. To set a new policy version as the active version, under **Policy version status**, select **Set the edited version as the active version for this policy**.
12. Choose **Save as new version**.

Configure the AWS IoT thing policy (AWS CLI)

1. List the principals for the core device's AWS IoT thing. Thing principals can be X.509 device certificates or other identifies. Run the following command, and replace *MyGreengrassCore* with the name of the core device.

   ```
   aws iot list-thing-principals --thing-name MyGreengrassCore
   ```

   The operation returns a response that lists the core device's thing principals.

   ```
   {
       "principals": [
           "arn:aws:iot:us-west-2:123456789012:cert/certificateId"
       ]
   }
   ```

2. Identify the core device's active certificate. Run the following command, and replace *certificateId* with the ID of each certificate from the previous step until you find the active certificate. The certificate ID is the hexadecimal string at the end of the certificate ARN. The `--query` argument specifies to output only the certificate's status.

   ```
   aws iot describe-certificate --certificate-id certificateId --query
    'certificateDescription.status'
   ```

   The operation returns the certificate status as a string. For example, if the certificate is active, this operation outputs `"ACTIVE"`.
3. List the AWS IoT policies that are attached to the certificate. Run the following command, and replace the certificate ARN with the ARN of the certificate.

   ```
   aws iot list-principal-policies --principal arn:aws:iot:us-
   west-2:123456789012:cert/certificateId
   ```

   The operation returns a response that lists the AWS IoT policies that are attached to the certificate.

```
{
    "policies": [
        {
            "policyName":
 "GreengrassTESCertificatePolicyMyGreengrassCoreTokenExchangeRoleAlias",
            "policyArn": "arn:aws:iot:us-west-2:123456789012:policy/
GreengrassTESCertificatePolicyMyGreengrassCoreTokenExchangeRoleAlias"
        },
        {
            "policyName": "GreengrassV2IoTThingPolicy",
            "policyArn": "arn:aws:iot:us-west-2:123456789012:policy/
GreengrassV2IoTThingPolicy"
        }
    ]
}
```

4. Choose the policy to view and update.

> **Note**
>
> If you used the AWS IoT Greengrass Core software installer to provision resources, you have two AWS IoT policies. We recommend that you choose the policy named **GreengrassV2IoTThingPolicy**, if it exists. Core devices that you create with the quick installer use this policy name by default. If you add permissions to this policy, you are also granting these permissions to other core devices that use this policy.

5. Get the policy's document. Run the following command, and replace *GreengrassV2IoTThingPolicy* with the name of the policy.

```
aws iot get-policy --policy-name GreengrassV2IoTThingPolicy
```

The operation returns a response that contains the policy's document and other information about the policy. The policy document is a JSON object serialized as a string.

```
{
    "policyName": "GreengrassV2IoTThingPolicy",
    "policyArn": "arn:aws:iot:us-west-2:123456789012:policy/
GreengrassV2IoTThingPolicy",
    "policyDocument": "{\
  \\"Version\\": \\"2012-10-17       \\",\
```

```
  \\"Statement\\": [\
    {\
      \\"Effect\\": \\"Allow\\",\
      \\"Action\\": [\
                \\"iot:Connect\\",\
                \\"iot:Publish\\",\
                \\"iot:Subscribe\\",\
                \\"iot:Receive\\",\
                \\"greengrass:*\\"\
],\
      \\"Resource\\": \\"*\\"\
    }\
  ]\
}",
    "defaultVersionId": "1",
    "creationDate": "2021-02-05T16:03:14.098000-08:00",
    "lastModifiedDate": "2021-02-05T16:03:14.098000-08:00",
    "generationId":
 "f19144b798534f52c619d44f771a354f1b957dfa2b850625d9f1d0fde530e75f"
}
```

6. Use an online converter or other tool to convert the policy document string to a JSON object, and then save it to a file named `iot-policy.json`.

   For example, if you have the jq tool installed, you can run the following command to get the policy document, convert it to a JSON object, and save the policy document as a JSON object.

   ```
   aws iot get-policy --policy-name GreengrassV2IoTThingPolicy --query
    'policyDocument' | jq fromjson >> iot-policy.json
   ```

7. Review the policy for the required permissions, and add any required permissions that are missing.

   For example, on a Linux-based system, you can run the following command to use GNU nano to open the file.

   ```
   nano iot-policy.json
   ```

   - `greengrass:PutCertificateAuthorities`
   - `greengrass:VerifyClientDeviceIdentity`
   - `greengrass:VerifyClientDeviceIoTCertificateAssociation`

- `greengrass:GetConnectivityInfo`
- `greengrass:UpdateConnectivityInfo` – (Optional) This permission is required to use the IP detector component, which reports the core device's network connectivity information to the AWS IoT Greengrass cloud service.
- `iot:GetThingShadow`, `iot:UpdateThingShadow`, and `iot:DeleteThingShadow` – (Optional) These permissions are required to use the shadow manager component to sync client device shadows with AWS IoT Core. This feature requires Greengrass nucleus v2.6.0 or later, shadow manager v2.2.0 or later, and MQTT bridge v2.2.0 or later.

8. Save the changes as a new version of the policy. Run the following command, and replace *GreengrassV2IoTThingPolicy* with the name of the policy.

```
aws iot create-policy-version --policy-name GreengrassV2IoTThingPolicy --policy-
document file://iot-policy.json --set-as-default
```

The operation returns a response similar to the following example if it succeeds.

```
{
    "policyArn": "arn:aws:iot:us-west-2:123456789012:policy/
GreengrassV2IoTThingPolicy",
    "policyDocument": "{\
  \\"Version\\": \\"2012-10-17       \\",\
  \\"Statement\\": [\
    {\
      \\"Effect\\": \\"Allow\\",\
      \\"Action\\": [\
\\t\\t\\"iot:Connect\\",\
\\t\\t\\"iot:Publish\\",\
\\t\\t\\"iot:Subscribe\\",\
\\t\\t\\"iot:Receive\\",\
\\t\\t\\"greengrass:*\\"\
      ],\
      \\"Resource\\": \\"*\\"\
    }\
  ]\
}",
    "policyVersionId": "2",
    "isDefaultVersion": true
}
```

Greengrass components for client device support

Important

The core device must run Greengrass nucleus v2.2.0 or later to support client devices.

To enable client devices to connect and communicate with a core device, you deploy the following Greengrass components to the core device:

- Client device auth (`aws.greengrass.clientdevices.Auth`)

  Deploy the client device auth component to authenticate client devices and authorize client device actions. This component allows your AWS IoT things to connect to a core device.

  This component requires some configuration to use it. You must specify groups of client devices and the operations that each group is authorized to perform, such as to connect and communicate over MQTT. For more information, see client device auth component configuration.

- MQTT 3.1.1 broker (Moquette) (`aws.greengrass.clientdevices.mqtt.Moquette`)

  Deploy the Moquette MQTT broker component to run a lightweight MQTT broker. The Moquette MQTT broker is compliant with MQTT 3.1.1 and includes local support for QoS 0, QoS 1, QoS 2, retained messages, last will messages, and persistent subscriptions.

  You aren't required to configure this component to use it. However, you can configure the port where this component operates the MQTT broker. By default, it uses port 8883.

- MQTT 5 broker (EMQX) (`aws.greengrass.clientdevices.mqtt.EMQX`)

  Note

  To use the EMQX MQTT 5 broker, you must use Greengrass nucleus v2.6.0 or later and client device auth v2.2.0 or later.

  Deploy the EMQX MQTT broker component to use MQTT 5.0 features in communication between client devices and the core device. The EMQX MQTT broker is compliant with MQTT 5.0 and includes support for session and message expiration intervals, user properties, shared subscriptions, topic aliases, and more.

You aren't required to configure this component to use it. However, you can configure the port where this component operates the MQTT broker. By default, it uses port 8883.

- MQTT bridge (`aws.greengrass.clientdevices.mqtt.Bridge`)

  (Optional) Deploy the MQTT bridge component to relay messages between client devices (local MQTT), local publish/subscribe, and AWS IoT Core MQTT. Configure this component to sync client devices with AWS IoT Core and interact with client devices from Greengrass components.

  This component requires configuration to use. You must specify the topic mappings where this component relays messages. For more information, see MQTT bridge component configuration.

- IP detector (`aws.greengrass.clientdevices.IPDetector`)

  (Optional) Deploy the IP detector component to automatically report the core device's MQTT broker endpoints to the AWS IoT Greengrass cloud service. You cannot use this component if you have a complex network setup, such as one where a router forwards the MQTT broker port to the core device.

  You aren't required to configure this component to use it.

- Shadow manager (`aws.greengrass.ShadowManager`)

  > **Note**
  >
  > To manage client device shadows, you must use Greengrass nucleus v2.6.0 or later, shadow manager v2.2.0 or later, and MQTT bridge v2.2.0 or later.

  (Optional) Deploy the shadow manager component to manage client device shadows on the core device. Greengrass components can get, update, and delete client device shadows to interact with client devices. You can also configure the shadow manager component to synchronize client device shadows with the AWS IoT Core cloud service.

  To use this component with client device shadows, you must configure the MQTT bridge component to relay messages between client devices and shadow manager, which uses local publish/subscribe. Otherwise, this component doesn't require configuration to use, but it does require configuration to sync device shadows.

> **Note**
>
> We recommend that you deploy only one MQTT broker component. The MQTT bridge and IP detector components work with only one MQTT broker component at a time. If you deploy multiple MQTT broker components, you must configure them to use different ports.

Configure cloud discovery (console)

You can use the AWS IoT Greengrass console to associate client devices, manage core device endpoints, and deploy components to enable client device support. For more information, see Step 2: Enable client device support.

Configure cloud discovery (AWS CLI)

You can use the AWS Command Line Interface (AWS CLI) to associate client devices, manage core device endpoints, and deploy components to enable client device support. For more information, see the following:

- Manage client device associations (AWS CLI)
- Manage core device endpoints
- AWS-provided client device components
- Create deployments

Associate client devices

To use cloud discovery, associate client devices with a core device so that they can discover the core device. Then, they can use the Greengrass discovery API to retrieve connectivity information and certificates for their associated core devices.

Likewise, disassociate client devices from a core device to stop them from discovering the core device.

Topics

- Manage client device associations (console)
- Manage client device associations (AWS CLI)
- Manage client device associations (API)

Manage client device associations (console)

You can use the AWS IoT Greengrass console to view, add, and delete client device associations.

To view client device associations for a core device (console)

1. Navigate to the AWS IoT Greengrass console.
2. Choose **Core devices**.
3. Choose the core device to manage.
4. On the core device's details page, choose the **Client devices** tab.
5. In the **Associated client devices** section, you can see which client devices (AWS IoT things) are associated with the core device.

To associate client devices with a core device (console)

1. Navigate to the AWS IoT Greengrass console.
2. Choose **Core devices**.
3. Choose the core device to manage.
4. On the core device's details page, choose the **Client devices** tab.
5. In the **Associated client devices** section, choose **Associate client devices**.
6. In the **Associate client devices with core device** modal, do the following for each client device to associate:
   a. Enter the name of the AWS IoT thing to associate as a client device.
   b. Choose **Add**.
7. Choose **Associate**.

   The client devices that you associated can now use the Greengrass discovery API to discover this core device.

To disassociate client devices from a core device (console)

1. Navigate to the AWS IoT Greengrass console.
2. Choose **Core devices**.
3. Choose the core device to manage.

4. On the core device's details page, choose the **Client devices** tab.
5. In the **Associated client devices** section, select each client device to disassociate.
6. Choose **Disassociate**.
7. In the confirmation modal, choose **Disassociate**.

   The client devices that you disassociated can no longer use the Greengrass discovery API to discover this core device.

Manage client device associations (AWS CLI)

You can use the AWS Command Line Interface (AWS CLI) to manage client device associations for a core device.

To view client device associations for a core device (AWS CLI)

- Use the following command: list-client-devices-associated-with-core-device.

To associate client devices with a core device (AWS CLI)

- Use the following command: batch-associate-client-device-with-core-device.

To disassociate client devices from a core device (AWS CLI)

- Use the following command: batch-disassociate-client-device-from-core-device.

Manage client device associations (API)

You can use the AWS API to manage client device associations for a core device.

To view client device associations for a core device (AWS API)

- Use the following operation: ListClientDevicesAssociatedWithCoreDevice.

To associate client devices with a core device (AWS API)

- Use the following operation: BatchAssociateClientDeviceWithCoreDevice.

To disassociate client devices from a core device (AWS API)

- Use the following operation: BatchDisassociateClientDeviceFromCoreDevice.

Authenticating clients while offline

With *offline authentication* you can configure your AWS IoT Greengrass Core device so that client devices can connect to a core device, even when the core device isn't connected to the cloud. When you use offline authentication, your Greengrass devices can continue to work in a partially offline environment.

To use offline authentication for a client device with a connection to the cloud, you need the following:

- An AWS IoT Greengrass Core device with the Client device auth component deployed. You must use version 2.3.0 or greater for offline authentication.
- A cloud connection for the core device during the initial connection of client devices.

Storing client credentials

When a client device connects to a core device for the first time, the core device calls the AWS IoT Greengrass service. When called, Greengrass validates the client device's registration as an AWS IoT thing. It also validates that the device has a valid certificate. The core device then stores this information locally.

The next time that the device connects, the Greengrass core device attempts to validate the client device with the AWS IoT Greengrass service. If it can't connect to AWS IoT Greengrass, the core device uses its locally stored device information to validate the client device.

You can configure the length of time that the Greengrass core device stores credentials. You can set the timeout from one minute to 2,147,483,647 minutes by setting the `clientDeviceTrustDurationMinutes` configuration option in the client device auth component configuration. The default is one minute, which effectively turns off offline authentication. When you set this timeout, we recommend that you consider your security needs. You should also consider how long you expect core devices to run while disconnected from the cloud.

The core device updates its credential storage at three times:

1. When a device connects to the core device for the first time.
2. If the core device is connected to the cloud, when a client device reconnects to the core device.
3. If the core device is connected to the cloud, once a day to refresh the entire credential store.

When the Greengrass core device refreshes its credential store, it uses the ListClientDevicesAssociatedWithCoreDevice operation. Greengrass only refreshes the devices returned by this operation. To associate a client device with a core device, see Associate client devices.

To use the `ListClientDevicesAssociatedWithCoreDevice` operation, you must add permission for the operation to the AWS Identity and Access Management (IAM) role associated with the AWS account that runs AWS IoT Greengrass. For more information, see Authorize core devices to interact with AWS services.

Manage core device endpoints

When you use cloud discovery, you store MQTT broker endpoints for core devices in the AWS IoT Greengrass cloud service. Client devices connect to AWS IoT Greengrass to retrieve these endpoints and other information for their associated core devices.

For each core device, you can automatically or manually manage endpoints.

- **Automatically manage endpoints with IP detector**

  You can deploy the IP detector component to automatically manage core device endpoints for you if you have a non-complex network setup, such as where the client devices are on the same network as the core device. You can't use the IP detector component if the core device is behind a router that forwards the MQTT broker port to the core device, for example.

  The IP detector component is also useful if you deploy to thing groups, because it manages the endpoints for all core devices in the thing group. For more information, see Use IP detector to automatically manage endpoints.

- **Manually manage endpoints**

  If you can't use the IP detector component, you must manually manage core device endpoints. You can update these endpoints with the console or the API. For more information, see Manually manage endpoints.

Topics

- Use IP detector to automatically manage endpoints
- Manually manage endpoints

Use IP detector to automatically manage endpoints

If you have a simple network setup, such as the client devices on the same network as the core device, you can deploy the IP detector component to do the following:

- Monitor the Greengrass core device's local network connectivity information. This information includes the core device's network endpoints and the port where the MQTT broker operates.
- Report the core device's connectivity information to the AWS IoT Greengrass cloud service.

The IP detector component overwrites endpoints that you set manually.

> **Important**
>
> The core device's AWS IoT policy must allow the `greengrass:UpdateConnectivityInfo` permission to use the IP detector component. For more information, see AWS IoT policies for data plane operations and Configure the AWS IoT thing policy.

You can do either of the following to deploy the IP detector component:

- Use the **Configure discovery** page in the console. For more information, see Configure cloud discovery (console).
- Create and revise deployments to include the IP detector. You can use the console, AWS CLI, or AWS API to manage deployments. For more information, see Create deployments.

Deploy the IP detector component (console)

1. In the AWS IoT Greengrass console navigation menu, choose **Components**.
2. On the **Components** page, choose the **Public components** tab, and then choose **aws.greengrass.clientdevices.IPDetector**.
3. On the **aws.greengrass.clientdevices.IPDetector** page, choose **Deploy**.

4. From **Add to deployment**, choose an existing deployment to revise, or choose to create a new deployment, and then choose **Next**.
5. If you chose to create a new deployment, choose the target core device or thing group for the deployment. On the **Specify target** page, under **Deployment target**, choose a core device or thing group, and then choose **Next**.
6. On the **Select components** page, verify that the **aws.greengrass.clientdevices.IPDetector** component is selected, choose **Next**.
7. On the **Configure components** page, select **aws.greengrass.clientdevices.IPDetector**, and then do the following:

   a. Choose **Configure component**.

   b. In the **Configure aws.greengrass.clientdevices.IPDetector** modal, under **Configuration update**, in **Configuration to merge**, you can enter a configuration update to configure the IP detector component. You can specify any of the following configuration options:

   - `defaultPort` – (Optional) The MQTT broker port to report when this component detects IP addresses. You must specify this parameter if you configure the MQTT broker to use a different port than the default port 8883.
   - `includeIPv4LoopbackAddrs` – (Optional) You can enable this option to detect and report IPv4 loopback addresses. These are IP addresses, such as `localhost`, where a device can communicate with itself. Use this option in test environments where the core device and client device run on the same system.
   - `includeIPv4LinkLocalAddrs` – (Optional) You can enable this option to detect and report IPv4 link-local addresses. Use this option if the core device's network doesn't have Dynamic Host Configuration Protocol (DHCP) or statically assigned IP addresses.
   - `includeIPv6LoopbackAddrs` – (Optional) You can enable this option to detect and report IPv6 loopback addresses. These are IP addresses, such as `localhost`, where a device can communicate with itself. Use this option in test environments where the core device and client device run on the same system. You must set `includeIPv4Addrs` to `false` and `includeIPv6Addrs` to `true` to use this option. You must have IP detector v2.2.0 or later to use this option.
   - `includeIPv6LinkLocalAddrs` – (Optional) You can enable this option to detect and report IPv6 link-local addresses. Use this option if the core device's network doesn't have Dynamic Host Configuration Protocol (DHCP) or statically assigned IP addresses.

You must set `includeIPv4Addrs` to `false` and `includeIPv6Addrs` to `true` to use this option. You must have IP detector v2.2.0 or later to use this option.

- `includeIPv4Addrs` – (Optional) The default is set to true. You can enable this option to publish IPv4 addresses found on the core device. You must have IP detector v2.2.0 or later to use this option.
- `includeIPv6Addrs` – (Optional) You can enable this option to publish IPv6 addresses found on the core device. Set `includeIPv4Addrs` to `false` to use this option. You must have IP detector v2.2.0 or later to use this option.

The configuration update might look similar to the following example.

```
{
  "defaultPort": "8883",
  "includeIPv4LoopbackAddrs": false,
  "includeIPv4LinkLocalAddrs": false
}
```

c. Choose **Confirm** to close the modal, and then choose **Next**.

8. On the **Configure advanced settings** page, keep the default configuration settings, and choose **Next**.
9. On the **Review** page, choose **Deploy**.

   The deployment can take up to a minute to complete.

Deploy the IP detector component (AWS CLI)

To deploy the IP detector component, create a deployment document that includes `aws.greengrass.clientdevices.IPDetector` in the `components` object, and specify the configuration update for the component. Follow instructions in Create deployments to create a new deployment or revise an existing deployment.

You can specify any of the following options to configure the IP detector component when you create the deployment document:

- `defaultPort` – (Optional) The MQTT broker port to report when this component detects IP addresses. You must specify this parameter if you configure the MQTT broker to use a different port than the default port 8883.

- `includeIPv4LoopbackAddrs` – (Optional) You can enable this option to detect and report IPv4 loopback addresses. These are IP addresses, such as `localhost`, where a device can communicate with itself. Use this option in test environments where the core device and client device run on the same system.
- `includeIPv4LinkLocalAddrs` – (Optional) You can enable this option to detect and report IPv4 link-local addresses. Use this option if the core device's network doesn't have Dynamic Host Configuration Protocol (DHCP) or statically assigned IP addresses.
- `includeIPv6LoopbackAddrs` – (Optional) You can enable this option to detect and report IPv6 loopback addresses. These are IP addresses, such as `localhost`, where a device can communicate with itself. Use this option in test environments where the core device and client device run on the same system. You must set `includeIPv4Addrs` to `false` and `includeIPv6Addrs` to `true` to use this option. You must have IP detector v2.2.0 or later to use this option.
- `includeIPv6LinkLocalAddrs` – (Optional) You can enable this option to detect and report IPv6 link-local addresses. Use this option if the core device's network doesn't have Dynamic Host Configuration Protocol (DHCP) or statically assigned IP addresses. You must set `includeIPv4Addrs` to `false` and `includeIPv6Addrs` to `true` to use this option. You must have IP detector v2.2.0 or later to use this option.
- `includeIPv4Addrs` – (Optional) The default is set to true. You can enable this option to publish IPv4 addresses found on the core device. You must have IP detector v2.2.0 or later to use this option.
- `includeIPv6Addrs` – (Optional) You can enable this option to publish IPv6 addresses found on the core device. Set `includeIPv4Addrs` to `false` to use this option. You must have IP detector v2.2.0 or later to use this option.

The following example partial deployment document specifies to report port 8883 as the MQTT broker port.

```
{
  ...,
  "components": {
    ...,
    "aws.greengrass.clientdevices.IPDetector": {
      "componentVersion": "2.1.1",
      "configurationUpdate": {
        "merge": "{\"defaultPort\":\"8883\",}"
      }
```

```
    }
  }
}
```

Manually manage endpoints

You can manually manage MQTT broker endpoints for core devices.

Each MQTT broker endpoint has the following information:

Endpoint (`HostAddress`)

An IP address or DNS address where client devices can connect to an MQTT broker on the core device.

Port (`PortNumber`)

The port where the MQTT broker operates on the core device.

You can configure this port on the Moquette MQTT broker component, which defaults to use port 8883.

Metadata (`Metadata`)

Additional metadata to provide to client devices that connect to this endpoint.

Topics

- Manage endpoints (console)
- Manage endpoints (AWS CLI)
- Manage endpoints (API)

Manage endpoints (console)

You can use the AWS IoT Greengrass console to view, update, and remove endpoints for a core device.

To manage endpoints for a core device (console)

1. Navigate to the AWS IoT Greengrass console.
2. Choose **Core devices**.

3. Choose the core device to manage.
4. On the core device's details page, choose the **Client devices** tab.
5. In the **MQTT broker endpoints** section, you can see the core device's MQTT broker endpoints. Choose **Manage endpoints**.
6. In the **Manage endpoints** modal, add or remove MQTT broker endpoints for the core device.
7. Choose **Update**.

Manage endpoints (AWS CLI)

You can use the AWS Command Line Interface (AWS CLI) to manage endpoints for a core device.

> **Note**
>
> Because client device support in AWS IoT Greengrass V2 is backward compatible with AWS IoT Greengrass V1, you can use AWS IoT Greengrass V2 or AWS IoT Greengrass V1 API operations to manage core device endpoints.

To get endpoints for a core device (AWS CLI)

- Use either of the following commands:
  - greengrassv2: get-connectivity-info
  - greengrass: get-connectivity-info

To update endpoints for a core device (AWS CLI)

- Use either of the following commands:
  - greengrassv2: update-connectivity-info
  - greengrass: update-connectivity-info

Manage endpoints (API)

You can use the AWS API to manage endpoints for a core device.

> **Note**
>
> Because client device support in AWS IoT Greengrass V2 is backward compatible with AWS IoT Greengrass V1, you can use AWS IoT Greengrass V2 or AWS IoT Greengrass V1 API operations to manage core device endpoints.

To get endpoints for a core device (AWS API)

- Use either of the following operations:
  - V2: GetConnectivityInfo
  - V1: GetConnectivityInfo

To update endpoints for a core device (AWS API)

- Use either of the following operations:
  - V2: UpdateConnectivityInfo
  - V1: UpdateConnectivityInfo

Choose an MQTT broker

AWS IoT Greengrass provides options for you to choose which local MQTT broker to run on your core devices. Client devices connect to the MQTT broker that runs on a core device, so choose an MQTT broker that is compatible with the client devices that you want to connect.

> **Note**
>
> We recommend that you deploy only one MQTT broker component. The MQTT bridge and IP detector components work with only one MQTT broker component at a time. If you deploy multiple MQTT broker components, you must configure them to use different ports.

You can choose from the following MQTT brokers:

- **MQTT 3.1.1 broker (Moquette)** – `aws.greengrass.clientdevices.mqtt.Moquette`

Choose this option for a lightweight MQTT broker that is compliant with the MQTT 3.1.1 standard. The AWS IoT Core MQTT broker and AWS IoT Device SDK are also compliant with the MQTT 3.1.1 standard, so you can use these features to create an application that uses MQTT 3.1.1 across your devices and the AWS Cloud.

- **MQTT 5 broker (EMQX)** – `aws.greengrass.clientdevices.mqtt.EMQX`

  Choose this option to use MQTT 5 features in communication between core devices and client devices. This component uses more resources than the Moquette MQTT 3.1.1 broker, and on Linux core devices, it requires Docker.

  MQTT 5 is backward-compatible with MQTT 3.1.1, so you can connect client devices that use MQTT 3.1.1 to this broker. If you run the Moquette MQTT 3.1.1 broker, you can replace it with the EMQX MQTT 5 broker, and client devices can continue to connect and operate as usual.

- **Implement a custom broker**

  Choose this option to create a custom local broker component to communicate with client devices. You can create a custom local broker that uses a protocol other than MQTT. AWS IoT Greengrass provides a component SDK that you can use to authenticate and authorize client devices. For more information, see Use the AWS IoT Device SDK to communicate with the Greengrass nucleus, other components, and AWS IoT Core and Authenticate and authorize client devices.

Connecting client devices to an AWS IoT Greengrass Core device with an MQTT broker

When you use an MQTT broker on your AWS IoT Greengrass Core device, the device uses a *core device certificate authority (CA)* unique to the device to issue a certificate to the broker for making mutual TLS connections with clients.

AWS IoT Greengrass will autogenerate a core device CA, or you can provide your own. The core device CA is registered with AWS IoT Greengrass when the Client device auth component is connected. The autogenerated core device CA is persistent, the device will continue to use the same CA as long as the client device auth component is configured.

When the MQTT broker starts, it requests a certificate. The client device auth component issues an X.509 certificate using the core device CA. The certificate is rotated when the broker starts, when

the certificate expires, or when connectivity information such as the IP address changes. For more information, see Certificate rotation on the local MQTT broker.

To connect a client to the MQTT broker, you need the following:

- The client device must have the AWS IoT Greengrass Core device CA. You can get this CA through cloud discovery, or by providing the CA manually. For more information, see Using your own certificate authority.
- The fully-qualified domain name (FQDN) or IP address of the core device must be present in the broker certificate issued by the core device CA. You ensure this using the IP detector component or manually configuring the IP address. For more information, see Manage core device endpoints.
- The client device auth component must give the client device permission to connect to the Greengrass core device. For more information, see Client device auth.

Using your own certificate authority

If your client devices can't access the cloud to discover your core device, you can provide a *core device certificate authority (CA)*. Your Greengrass core device uses the core device CA to issue certificates for your MQTT broker. Once you configure the core device and provision your client device with its CA, your client devices can connect to the endpoint and verify the TLS handshake using the core device CA (own provided CA or autogenerated).

To configure the Client device auth component to use your core device CA, set the `certificateAuthority` configuration parameter when you deploy the component. You must provide the following details during configuration:

- The location of a core device CA certificate.
- The private key of the core device CA certificate.
- (Optional) The certificate chain to the root certificate if the core device CA is an intermediate CA.

If you provide a core device CA, AWS IoT Greengrass registers the CA with the cloud.

You can store your certificates in a hardware security module or on the file system. The following example shows a `certificateAuthority` configuration for a intermediate CA stored using HSM/TPM. Note that the certificate chain can only be stored on disk.

```
  "certificateAuthority": {
```

```
    "certificateUri": "pkcs11:object=CustomerIntermediateCA;type=cert",
    "privateKeyUri": "pkcs11:object=CustomerIntermediateCA;type=private"
    "certificateChainUri": "file:///home/ec2-user/creds/certificateChain.pem",
  }
```

In this example, the `certificateAuthority` configuration parameter configures the client device auth component to use an intermediate CA from the file system:

```
"certificateAuthority": {
    "certificateUri": "file:///home/ec2-user/creds/intermediateCA.pem",
    "privateKeyUri": "file:///home/ec2-user/creds/intermediateCA.privateKey.pem",
    "certificateChainUri": "file:///home/ec2-user/creds/certificateChain.pem",
  }
```

To connect the devices to your AWS IoT Greengrass Core device, do the following:

1. Create an intermediate certificate authority (CA) for the Greengrass core device using your organization's root CA. We recommend that you use an intermediate CA as a security best practice.
2. Provide the intermediate CA certificate, private key, and the certificate chain to your root CA to the Greengrass core device. For more information, see Client device auth. The intermediate CA becomes the core device CA for the Greengrass core device, and the device registers the CA with AWS IoT Greengrass.
3. Register the client device as an AWS IoT thing. For more information, see Create a thing object in the *AWS IoT Core Developer Guide*. Add the private key, public key, device certificate, and root CA certificate to your client device. How you add the information depends on your device and software.

Once you configure your device, you can use the certificate and public key chain to connect to the Greengrass core device. Your software is responsible for finding the core device endpoints. You can set the endpoint manually for the core device. For more information, see Manually manage endpoints.

Test client device communications

Client devices can use the AWS IoT Device SDK to discover, connect, and communicate with a core device. You can use the Greengrass discovery client in the AWS IoT Device SDK to use the Greengrass discovery API, which returns information about core devices to which a client device

can connect. The API response includes MQTT broker endpoints to connect and certificates to use to verify the identity of each core device. Then, the client device can try each endpoint until it successfully connects to a core device.

Client devices can discover only core devices to which you associate them. Before you test communications between a client device and a core device, you must associate the client device to the core device. For more information, see Associate client devices.

The Greengrass discovery API returns the core device MQTT broker endpoints that you specify. You can use the IP detector component to manage these endpoints for you, or you can manually manage them for each core device. For more information, see Manage core device endpoints.

> **Note**
>
> To use the Greengrass discovery API, a client device must have the `greengrass:Discover` permission. For more information, see Minimal AWS IoT policy for client devices.

The AWS IoT Device SDK is available in multiple programming languages. For more information, see AWS IoT Device SDKs in the *AWS IoT Core Developer Guide*.

Topics

- Test communications (Python)
- Test communications (C++)
- Test communications (JavaScript)
- Test communications (Java)

Test communications (Python)

In this section, you use Greengrass discovery sample in the AWS IoT Device SDK v2 for Python to test communications between a client device and a core device.

> **Important**
>
> To use the AWS IoT Device SDK v2 for Python, a device must run Python 3.6 or later.

To test communications (AWS IoT Device SDK v2 for Python)

1. Download and install the AWS IoT Device SDK v2 for Python to the AWS IoT thing to connect as a client device.

   On the client device, do the following:

   a. Clone the AWS IoT Device SDK v2 for Python repository to download it.

   ```
   git clone https://github.com/aws/aws-iot-device-sdk-python-v2.git
   ```

   b. Install the AWS IoT Device SDK v2 for Python.

   ```
   python3 -m pip install --user ./aws-iot-device-sdk-python-v2
   ```

2. Change to the samples folder in the AWS IoT Device SDK v2 for Python.

   ```
   cd aws-iot-device-sdk-python-v2/samples/greengrass
   ```

3. Run the sample Greengrass discovery application. This application expects arguments that specify the client device thing name, the MQTT topic and message to use, and the certificates that authenticate and secure the connection. The following example sends a Hello World message to the `clients/`*`MyClientDevice1`*`/hello/world` topic.

   - Replace *`MyClientDevice1`* with the client device's thing name.
   - Replace *`~/certs/AmazonRootCA1.pem`* with the path to the Amazon root CA certificate on the client device.
   - Replace *`~/certs/device.pem.crt`* with the path to the device certificate on the client device.
   - Replace *`~/certs/private.pem.key`* with the path to the private key file on the client device.
   - Replace *`us-east-1`* with the AWS Region where your client device and core device operate.

   ```
   python3 basic_discovery.py \\
     --thing_name MyClientDevice1 \\
     --topic 'clients/MyClientDevice1/hello/world' \\
     --message 'Hello World!' \\
     --ca_file ~/certs/AmazonRootCA1.pem \\
     --cert ~/certs/device.pem.crt \\
   ```

```
  --key ~/certs/private.pem.key \\
  --region us-east-1 \\
  --verbosity Warn
```

The discovery sample application sends the message 10 times and disconnects. It also subscribes to the same topic where it publishes messages. If the output indicates that the application received MQTT messages on the topic, the client device can successfully communicate with the core device.

```
Performing greengrass discovery...
awsiot.greengrass_discovery.DiscoverResponse(gg_groups=[awsiot.greengrass_discovery.GGGroup
coreDevice-MyGreengrassCore',
 cores=[awsiot.greengrass_discovery.GGCore(thing_arn='arn:aws:iot:us-
east-1:123456789012:thing/MyGreengrassCore',
 connectivity=[awsiot.greengrass_discovery.ConnectivityInfo(id='203.0.113.0',
 host_address='203.0.113.0', metadata='', port=8883)])],
 certificate_authorities=['-----BEGIN CERTIFICATE-----\
MIICiT...EXAMPLE=\
-----END CERTIFICATE-----\
'])])
Trying core arn:aws:iot:us-east-1:123456789012:thing/MyGreengrassCore at host
 203.0.113.0 port 8883
Connected!
Published topic clients/MyClientDevice1/hello/world: {"message": "Hello World!",
 "sequence": 0}

Publish received on topic clients/MyClientDevice1/hello/world
b'{"message": "Hello World!", "sequence": 0}'
Published topic clients/MyClientDevice1/hello/world: {"message": "Hello World!",
 "sequence": 1}

Publish received on topic clients/MyClientDevice1/hello/world
b'{"message": "Hello World!", "sequence": 1}'

...

Published topic clients/MyClientDevice1/hello/world: {"message": "Hello World!",
 "sequence": 9}

Publish received on topic clients/MyClientDevice1/hello/world
b'{"message": "Hello World!", "sequence": 9}'
```

If the application outputs an error instead, see Troubleshooting Greengrass discovery issues.

You can also view the Greengrass logs on the core device to verify if the client device successfully connects and sends messages. For more information, see Monitor AWS IoT Greengrass logs.

Test communications (C++)

In this section, you use Greengrass discovery sample in the AWS IoT Device SDK v2 for C++ to test communications between a client device and a core device.

To build the AWS IoT Device SDK v2 for C++, a device must have the following tools:

- C++ 11 or later
- CMake 3.1 or later
- One of the following compilers:
  - GCC 4.8 or later
  - Clang 3.9 or later
  - MSVC 2015 or later

To test communications (AWS IoT Device SDK v2 for C++)

1. Download and build the AWS IoT Device SDK v2 for C++ to the AWS IoT thing to connect as a client device.

   On the client device, do the following:

   a. Create a folder for the AWS IoT Device SDK v2 for C++ workspace, and change to it.

   ```
   cd
   mkdir iot-device-sdk-cpp
   cd iot-device-sdk-cpp
   ```

   b. Clone the AWS IoT Device SDK v2 for C++ repository to download it. The `--recursive` flag specifies to download submodules.

   ```
   git clone --recursive https://github.com/aws/aws-iot-device-sdk-cpp-v2.git
   ```

c. Create a folder for the AWS IoT Device SDK v2 for C++ build output, and change to it.

```
mkdir aws-iot-device-sdk-cpp-v2-build
cd aws-iot-device-sdk-cpp-v2-build
```

d. Build the AWS IoT Device SDK v2 for C++.

```
cmake -DCMAKE_INSTALL_PREFIX="~/iot-device-sdk-cpp" -
DCMAKE_BUILD_TYPE="Release" ../aws-iot-device-sdk-cpp-v2
cmake --build . --target install
```

2. Build the Greengrass discovery sample application in the AWS IoT Device SDK v2 for C++. Do the following:

a. Change to the Greengrass discovery sample folder in the AWS IoT Device SDK v2 for C++.

```
cd ../aws-iot-device-sdk-cpp-v2/samples/greengrass/basic_discovery
```

b. Create a folder for the Greengrass discovery sample build output, and change to it.

```
mkdir build
cd build
```

c. Build the Greengrass discovery sample application.

```
cmake -DCMAKE_PREFIX_PATH="~/iot-device-sdk-cpp" -
DCMAKE_BUILD_TYPE="Release" ..
cmake --build . --config "Release"
```

3. Run the sample Greengrass discovery application. This application expects arguments that specify the client device thing name, the MQTT topic to use, and the certificates that authenticate and secure the connection. The following example subscribes to the `clients/`*`MyClientDevice1`*`/hello/world` topic and publishes a message that you enter on the command line to the same topic.

- Replace *`MyClientDevice1`* with the client device's thing name.
- Replace *`~/certs/AmazonRootCA1.pem`* with the path to the Amazon root CA certificate on the client device.
- Replace *`~/certs/device.pem.crt`* with the path to the device certificate on the client device.

- Replace *~/certs/private.pem.key* with the path to the private key file on the client device.
- Replace *us-east-1* with the AWS Region where your client device and core device operate.

```
./basic-discovery \
  --thing_name MyClientDevice1 \
  --topic 'clients/MyClientDevice1/hello/world' \
  --ca_file ~/certs/AmazonRootCA1.pem \
  --cert ~/certs/device.pem.crt \
  --key ~/certs/private.pem.key \
  --region us-east-1
```

The discovery sample application subscribes to the topic and prompts you to enter a message to publish.

```
Connecting to group greengrassV2-coreDevice-MyGreengrassCore with thing arn
 arn:aws:iot:us-east-1:123456789012:thing/MyGreengrassCore, using endpoint
 203.0.113.0:8883
Connected to group greengrassV2-coreDevice-MyGreengrassCore, using connection to
 203.0.113.0:8883
Successfully subscribed to clients/MyClientDevice1/hello/world
Enter the message you want to publish to topic clients/MyClientDevice1/hello/world
 and press enter. Enter 'exit' to exit this program.
```

If the application outputs an error instead, see Troubleshooting Greengrass discovery issues.

4. Enter a message, such as **`Hello World!`**.

```
Enter the message you want to publish to topic clients/MyClientDevice1/hello/world
 and press enter. Enter 'exit' to exit this program.
Hello World!
```

If the output indicates that the application received the MQTT message on the topic, the client device can successfully communicate with the core device.

```
Operation on packetId 2 Succeeded
Publish received on topic clients/MyClientDevice1/hello/world
Message:
Hello World!
```

You can also view the Greengrass logs on the core device to verify if the client device successfully connects and sends messages. For more information, see Monitor AWS IoT Greengrass logs.

Test communications (JavaScript)

In this section, you use Greengrass discovery sample in the AWS IoT Device SDK v2 for JavaScript to test communications between a client device and a core device.

> **Important**
>
> To use the AWS IoT Device SDK v2 for JavaScript, a device must run Node v10.0 or later.

To test communications (AWS IoT Device SDK v2 for JavaScript)

1. Download and install the AWS IoT Device SDK v2 for JavaScript to the AWS IoT thing to connect as a client device.

   On the client device, do the following:

   a. Clone the AWS IoT Device SDK v2 for JavaScript repository to download it.

   ```
   git clone https://github.com/aws/aws-iot-device-sdk-js-v2.git
   ```

   b. Install the AWS IoT Device SDK v2 for JavaScript.

   ```
   cd aws-iot-device-sdk-js-v2
   npm install
   ```

2. Change to the Greengrass discovery sample folder in the AWS IoT Device SDK v2 for JavaScript.

   ```
   cd samples/node/greengrass/basic_discovery
   ```

3. Install the Greengrass discovery sample application.

   ```
   npm install
   ```

4. Run the sample Greengrass discovery application. This application expects arguments that specify the client device thing name, the MQTT topic and message to use, and the certificates that authenticate and secure the connection. The following example sends a Hello World message to the `clients/MyClientDevice1/hello/world` topic.

   - Replace *MyClientDevice1* with the client device's thing name.
   - Replace *~/certs/AmazonRootCA1.pem* with the path to the Amazon root CA certificate on the client device.
   - Replace *~/certs/device.pem.crt* with the path to the device certificate on the client device.
   - Replace *~/certs/private.pem.key* with the path to the private key file on the client device.
   - Replace *us-east-1* with the AWS Region where your client device and core device operate.

   ```
   node dist/index.js \
     --thing_name MyClientDevice1 \
     --topic 'clients/MyClientDevice1/hello/world' \
     --message 'Hello World!' \
     --ca_file ~/certs/AmazonRootCA1.pem \
     --cert ~/certs/device.pem.crt \
     --key ~/certs/private.pem.key \
     --region us-east-1 \
     --verbose warn
   ```

   The discovery sample application sends the message 10 times and disconnects. It also subscribes to the same topic where it publishes messages. If the output indicates that the application received MQTT messages on the topic, the client device can successfully communicate with the core device.

   ```
   Discovery Response:
   {"gg_groups":[{"gg_group_id":"greengrassV2-coreDevice-
   MyGreengrassCore","cores":[{"thing_arn":"arn:aws:iot:us-
   east-1:123456789012:thing/MyGreengrassCore","connectivity":
   [{"id":"203.0.113.0","host_address":"203.0.113.0","port":8883,"metadata":""}]}],"certificat
   ["-----BEGIN CERTIFICATE-----\nMIICiT...EXAMPLE=\n-----END CERTIFICATE-----\n"]}]}
   Trying
    endpoint={"id":"203.0.113.0","host_address":"203.0.113.0","port":8883,"metadata":""}
   ```

```
[WARN] [2021-06-12T00:46:45Z] [00007f90c0e8d700] [socket] - id=0x7f90b8018710
 fd=26: setsockopt() for NO_SIGNAL failed with errno 92. If you are having SIGPIPE
 signals thrown, you may want to install a signal trap in your application layer.
Connected to
 endpoint={"id":"203.0.113.0","host_address":"203.0.113.0","port":8883,"metadata":""}
Publish received. topic:"clients/MyClientDevice1/hello/world" dup:false qos:0
 retain:false
{"message":"Hello World!","sequence":1}
Publish received. topic:"clients/MyClientDevice1/hello/world" dup:false qos:0
 retain:false
{"message":"Hello World!","sequence":2}
Publish received. topic:"clients/MyClientDevice1/hello/world" dup:false qos:0
 retain:false
{"message":"Hello World!","sequence":3}
Publish received. topic:"clients/MyClientDevice1/hello/world" dup:false qos:0
 retain:false
{"message":"Hello World!","sequence":4}
Publish received. topic:"clients/MyClientDevice1/hello/world" dup:false qos:0
 retain:false
{"message":"Hello World!","sequence":5}
Publish received. topic:"clients/MyClientDevice1/hello/world" dup:false qos:0
 retain:false
{"message":"Hello World!","sequence":6}
Publish received. topic:"clients/MyClientDevice1/hello/world" dup:false qos:0
 retain:false
{"message":"Hello World!","sequence":7}
Publish received. topic:"clients/MyClientDevice1/hello/world" dup:false qos:0
 retain:false
{"message":"Hello World!","sequence":8}
Publish received. topic:"clients/MyClientDevice1/hello/world" dup:false qos:0
 retain:false
{"message":"Hello World!","sequence":9}
Publish received. topic:"clients/MyClientDevice1/hello/world" dup:false qos:0
 retain:false
{"message":"Hello World!","sequence":10}
Complete!
```

If the application outputs an error instead, see Troubleshooting Greengrass discovery issues.

You can also view the Greengrass logs on the core device to verify if the client device successfully connects and sends messages. For more information, see Monitor AWS IoT Greengrass logs.

Test communications (Java)

In this section, you use Greengrass discovery sample in the AWS IoT Device SDK v2 for Java to test communications between a client device and a core device.

> **Important**
>
> To build the AWS IoT Device SDK v2 for Java, a device must have the following tools:
>
> - Java 8 or later, with `JAVA_HOME` pointing to the Java folder.
> - Apache Maven

To test communications (AWS IoT Device SDK v2 for Java)

1. Download and build the AWS IoT Device SDK v2 for Java to the AWS IoT thing to connect as a client device.

   On the client device, do the following:

   a. Clone the AWS IoT Device SDK v2 for Java repository to download it.

   ```
   git clone https://github.com/aws/aws-iot-device-sdk-java-v2.git
   ```

   b. Change to the AWS IoT Device SDK v2 for Java folder.

   c. Build the AWS IoT Device SDK v2 for Java.

   ```
   cd aws-iot-device-sdk-java-v2
   mvn versions:use-latest-versions -Dincludes="software.amazon.awssdk.crt*"
   mvn clean install
   ```

2. Run the sample Greengrass discovery application. This application expects arguments that specify the client device thing name, the MQTT topic to use, and the certificates that authenticate and secure the connection. The following example subscribes to the `clients/MyClientDevice1/hello/world` topic and publishes a message that you enter on the command line to the same topic.

   - Replace both instances of *`MyClientDevice1`* with the client device's thing name.
   - Replace *`$HOME/certs/AmazonRootCA1.pem`* with the path to the Amazon root CA certificate on the client device.

- Replace *$HOME/certs/device.pem.crt* with the path to the device certificate on the client device.
- Replace *$HOME/certs/private.pem.key* with the path to the private key file on the client device.
- Replace *us-east-1* with the AWS Region where your client device and core device operate.

```
DISCOVERY_SAMPLE_ARGS="--thing_name MyClientDevice1 \
  --topic 'clients/MyClientDevice1/hello/world' \
  --ca_file $HOME/certs/AmazonRootCA1.pem \
  --cert $HOME/certs/device.pem.crt \
  --key $HOME/certs/private.pem.key \
  --region us-east-1"

mvn exec:java -pl samples/Greengrass/Discovery \
  -Dexec.mainClass=greengrass.BasicDiscovery \
  -Dexec.args="$DISCOVERY_SAMPLE_ARGS"
```

The discovery sample application subscribes to the topic and prompts you to enter a message to publish.

```
Connecting to group ID greengrassV2-coreDevice-MyGreengrassCore, with thing
 arn arn:aws:iot:us-east-1:123456789012:thing/MyGreengrassCore, using endpoint
 203.0.113.0:8883
Started a clean session
Enter the message you want to publish to topic clients/MyClientDevice1/hello/world
 and press Enter. Type 'exit' or 'quit' to exit this program:
```

If the application outputs an error instead, see Troubleshooting Greengrass discovery issues.

3. Enter a message, such as **`Hello World!`**.

```
Enter the message you want to publish to topic clients/MyClientDevice1/hello/world
 and press Enter. Type 'exit' or 'quit' to exit this program:
Hello World!
```

If the output indicates that the application received the MQTT message on the topic, the client device can successfully communicate with the core device.

```
Message received on topic clients/MyClientDevice1/hello/world: Hello World!
```

You can also view the Greengrass logs on the core device to verify if the client device successfully connects and sends messages. For more information, see Monitor AWS IoT Greengrass logs.

Greengrass discovery RESTful API

AWS IoT Greengrass provides the `Discover` API operation that client devices can use to identify Greengrass core devices where they can connect. Client devices use this data plane operation to retrieve information required to connect to Greengrass core devices where you associate them with the BatchAssociateClientDeviceWithCoreDevice API operation. When a client device comes online, it can connect to the AWS IoT Greengrass cloud service and use the discovery API to find:

- The IP address and port for each associated Greengrass core device.
- The core device CA certificate, which client devices can use to authenticate the Greengrass core device.

> **Note**
>
> Client devices can also use the discovery client in the AWS IoT Device SDK to discover connectivity information for Greengrass core devices. The discovery client uses the discovery API. For more information, see the following:
>
> - Test client device communications
> - Greengrass Discovery RESTful API in the *AWS IoT Greengrass Version 1 Developer Guide*.

To use this API operation, send HTTP requests to the discovery API on the Greengrass data plane endpoint. This API endpoint has the following format.

```
https://greengrass-ats.iot.region.amazonaws.com:port/greengrass/discover/thing/thing-name
```

For a list of supported AWS Regions and endpoints for the AWS IoT Greengrass discovery API, see AWS IoT Greengrass V2 endpoints and quotas in the *AWS General Reference*. This API operation is

available only on the Greengrass data plane endpoint. The control plane endpoint that you use to manage components and deployments is different from the data plane endpoint.

> **Note**
>
> The discovery API is the same for AWS IoT Greengrass V1 and AWS IoT Greengrass V2. If you have client devices that connect to an AWS IoT Greengrass V1 core, you can connect them to AWS IoT Greengrass V2 core devices without changing the code on the client devices. For more information, see Greengrass Discovery RESTful API in the *AWS IoT Greengrass Version 1 Developer Guide*.

Topics

- Discovery authentication and authorization
- Request
- Response
- Test the discovery API with cURL

Discovery authentication and authorization

To use the discovery API to retrieve connectivity information, a client device must use TLS mutual authentication with an X.509 client certificate to authenticate. For more information, see X.509 client certificates in the *AWS IoT Core Developer Guide*.

A client device must also have permission to perform the `greengrass:Discover` action. The following example AWS IoT policy allows an AWS IoT thing named `MyClientDevice1` to perform `Discover` for itself.

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "greengrass:Discover",
      "Resource": [
```

```
        "arn:aws:iot:us-west-2:123456789012:thing/MyClientDevice1"
      ]
    }
  ]
}
```

Important

Thing policy variables (`iot:Connection.Thing.*`) aren't supported for in AWS IoT policies for core devices or Greengrass data plane operations. Instead, you can use a wildcard that matches multiple devices that have similar names. For example, you can specify `MyGreengrassDevice*` to match `MyGreengrassDevice1`, `MyGreengrassDevice2`, and so on.

For more information, see AWS IoT Core policies in the *AWS IoT Core Developer Guide*.

Request

The request contains the standard HTTP headers and is sent to the Greengrass discovery endpoint, as shown in the following examples.

The port number depends on whether the core device is configured to send HTTPS traffic over port 8443 or port 443. For more information, see the section called "Connect on port 443 or through a network proxy".

Note

These examples use the Amazon Trust Services (ATS) endpoint, which works with the recommended ATS root CA certificates. Endpoints must match the root CA certificate type.

Port 8443

```
HTTP GET https://greengrass-ats.iot.region.amazonaws.com:8443/greengrass/discover/
thing/thing-name
```

Port 443

```
HTTP GET https://greengrass-ats.iot.region.amazonaws.com:443/greengrass/discover/
thing/thing-name
```

> **Note**
>
> Clients that connect on port 443 must implement the Application Layer Protocol Negotiation (ALPN) TLS extension and pass `x-amzn-http-ca` as the `ProtocolName` in the `ProtocolNameList`. For more information, see Protocols in the *AWS IoT Developer Guide*.

Response

Upon success, the response header includes the HTTP 200 status code and the response body contains the discover response document.

> **Note**
>
> Because AWS IoT Greengrass V2 uses the same discovery API as AWS IoT Greengrass V1, the response organizes information according to AWS IoT Greengrass V1 concepts, such as Greengrass groups. The response contains a list of Greengrass groups. In AWS IoT Greengrass V2, each core device is in its own group, where the group contains only that core device and its connectivity information.

Example discover response documents

The following document shows the response for a client device that is associated to one Greengrass core device. The core device has one endpoint and one CA certificate.

```
{
  "GGGroups": [
    {
      "GGGroupId": "greengrassV2-coreDevice-core-device-01-thing-name",
      "Cores": [
        {
          "thingArn": "core-device-01-thing-arn",
```

```
          "Connectivity": [
            {
              "id": "core-device-01-connection-id",
              "hostAddress": "core-device-01-address",
              "portNumber": core-device-01-port,
              "metadata": "core-device-01-description"
            }
          ]
        }
      ],
      "CAs": [
        "-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----"
      ]
    }
  ]
}
```

The following document shows the response for a client device that is associated to two core devices. The core devices have multiple endpoints and multiple group CA certificates.

```
{
  "GGGroups": [
    {
      "GGGroupId": "greengrassV2-coreDevice-core-device-01-thing-name",
      "Cores": [
        {
          "thingArn": "core-device-01-thing-arn",
          "Connectivity": [
            {
              "id": "core-device-01-connection-id",
              "hostAddress": "core-device-01-address",
              "portNumber": core-device-01-port,
              "metadata": "core-device-01-connection-1-description"
            },
            {
              "id": "core-device-01-connection-id-2",
              "hostAddress": "core-device-01-address-2",
              "portNumber": core-device-01-port-2,
              "metadata": "core-device-01-connection-2-description"
            }
          ]
        }
      ],
```

```
      "CAs": [
        "-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----",
        "-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----",
        "-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----"
      ]
    },
    {
      "GGGroupId": "greengrassV2-coreDevice-core-device-02-thing-name",
      "Cores": [
        {
          "thingArn":"core-device-02-thing-arn",
          "Connectivity" : [
            {
              "id": "core-device-02-connection-id",
              "hostAddress": "core-device-02-address",
              "portNumber": core-device-02-port,
              "metadata": "core-device-02-connection-1-description"
            }
          ]
        }
      ],
      "CAs": [
        "-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----",
        "-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----",
        "-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----"
      ]
    }
  ]
}
```

Test the discovery API with cURL

If you have cURL installed, you can test the discovery API. The following example specifies a client device's certificates to authenticate a request to the Greengrass discovery API endpoint.

```
curl -i \
  --cert 1a23bc4d56.cert.pem \
  --key 1a23bc4d56.private.key \
  https://greengrass-ats.iot.us-west-2.amazonaws.com:8443/greengrass/discover/
thing/MyClientDevice1
```

> **Note**
>
> The `-i` argument specifies to output HTTP response headers. You can use this option to help identify errors.

If the request succeeds, this command outputs a response similar to the following example.

```
{
  "GGGroups": [
    {
      "GGGroupId": "greengrassV2-coreDevice-MyGreengrassCore",
      "Cores": [
        {
          "thingArn": "arn:aws:iot:us-west-2:123456789012:thing/MyGreengrassCore",
          "Connectivity": [
            {
              "Id": "AUTOIP_192.168.1.4_1",
              "HostAddress": "192.168.1.5",
              "PortNumber": 8883,
              "Metadata": ""
            }
          ]
        }
      ],
      "CAs": [
        "-----BEGIN CERTIFICATE-----\ncert-contents\n-----END CERTIFICATE-----\n"
      ]
    }
  ]
}
```

If the command outputs an error, see Troubleshooting Greengrass discovery issues.

Relay MQTT messages between client devices and AWS IoT Core

You can relay MQTT messages and other data between client devices and AWS IoT Core. Client devices connect to the MQTT broker component that runs on the core device. By default, core devices don't relay MQTT messages or data between client devices and AWS IoT Core. Client devices can communicate only with each other over MQTT by default.

To relay MQTT messages between client devices and AWS IoT Core, configure the MQTT bridge component to do the following:

- Relay messages from client devices to AWS IoT Core.
- Relay messages from AWS IoT Core to client devices.

> **Note**
>
> The MQTT bridge uses QoS 1 to publish and subscribe to AWS IoT Core, even when a client device uses QoS 0 to publish and subscribe to the local MQTT broker. As a result, you might observe additional latency when you relay MQTT messages from client devices on the local MQTT broker to AWS IoT Core. For more information about MQTT configuration on core devices, see Configure MQTT timeouts and cache settings.

Topics

- Configure and deploy the MQTT bridge component
- Relay MQTT messages

Configure and deploy the MQTT bridge component

The MQTT bridge component consumes a list of topic mappings that each specify a message source and a message destination. To relay messages between client devices and AWS IoT Core, deploy the MQTT bridge component, and specify each source and destination topic in the component configuration.

To deploy the MQTT bridge component to a core device or group of core devices, create a deployment that includes the `aws.greengrass.clientdevices.mqtt.Bridge` component. Specify the topic mappings, `mqttTopicMapping`, in the MQTT bridge component configuration in the deployment.

The following example defines a deployment that configures the MQTT bridge component to relay messages on topics that match the `clients/+/hello/world` topic filter from client devices to AWS IoT Core. The `merge` configuration update requires a serialized JSON object. For more information, see Update component configurations.

Console

```
{
  "mqttTopicMapping": {
    "HelloWorldIotCore": {
      "topic": "clients/+/hello/world",
      "source": "LocalMqtt",
      "target": "IotCore"
    }
  }
}
```

AWS CLI

```
{
  "components": {
    "aws.greengrass.clientdevices.mqtt.Bridge": {
      "version": "2.0.0",
      "configurationUpdate": {
        "merge": "{\"mqttTopicMapping\":{\"HelloWorldIotCore\":{\"topic\":\"clients/
+/hello/world\",\"source\":\"LocalMqtt\",\"target\":\"IotCore\"}}}"
      }
    }
    ...
  }
}
```

Relay MQTT messages

To relay MQTT messages between client devices and AWS IoT Core, configure and deploy the MQTT Bridge component and specify the topics to relay.

Example: Relay messages on a topic from client devices to AWS IoT Core

The following MQTT bridge component configuration specifies relaying messages on topics that match the `clients/+/hello/world/event` topic filter from client devices to AWS IoT Core.

```
{
  "mqttTopicMapping": {
    "HelloWorldEvent": {
      "topic": "clients/+/hello/world/event",
```

```
      "source": "LocalMqtt",
      "target": "IotCore"
    }
  }
}
```

Example: Relay messages on a topic from AWS IoT Core to client devices

The following MQTT bridge component configuration specifies relaying messages on topics that match the `clients/+/hello/world/event/response` topic filter from AWS IoT Core to client devices.

```
{
  "mqttTopicMapping": {
    "HelloWorldEventConfirmation": {
      "topic": "clients/+/hello/world/event/response",
      "source": "IotCore",
      "target": "LocalMqtt"
    }
  }
}
```

Interact with client devices in components

You can develop custom Greengrass components that interact with client devices connected to a core device. For example, you can develop components that do the following:

- Act on MQTT messages from client devices and send data to AWS Cloud destinations.
- Send MQTT messages to client devices to initiate actions.

Client devices connect to and communicate with a core device through the MQTT broker component that runs on the core device. By default, client devices can communicate only with each other over MQTT, and Greengrass components can't receive these MQTT messages or send messages to client devices.

Greengrass components use the local publish/subscribe interface to communicate on a core device. To communicate with client devices in Greengrass components, configure the MQTT bridge component to do the following:

- Relay MQTT messages from client devices to local publish/subscribe.

- Relay MQTT messages from local publish/subscribe to client devices.

You can also interact with client device shadows in Greengrass components. For more information, see Interact with and sync client device shadows.

Topics

- Configure and deploy the MQTT bridge component
- Receive MQTT messages from client devices
- Send MQTT messages to client devices

Configure and deploy the MQTT bridge component

The MQTT bridge component consumes a list of topic mappings that each specify a message source and a message destination. To communicate with client devices, deploy the MQTT bridge component, and specify each source and destination topic in the component configuration.

To deploy the MQTT bridge component to a core device or group of core devices, create a deployment that includes the `aws.greengrass.clientdevices.mqtt.Bridge` component. Specify the topic mappings, `mqttTopicMapping`, in the MQTT bridge component configuration in the deployment.

The following example defines a deployment that configures the MQTT bridge component to relay the `clients/MyClientDevice1/hello/world` topic from client devices to local publish/subscribe broker. The `merge` configuration update requires a serialized JSON object. For more information, see Update component configurations.

Console

```
{
  "mqttTopicMapping": {
    "HelloWorldPubsub": {
      "topic": "clients/MyClientDevice1/hello/world",
      "source": "LocalMqtt",
      "target": "Pubsub"
    }
  }
}
```

AWS CLI

```
{
  "components": {
    "aws.greengrass.clientdevices.mqtt.Bridge": {
      "version": "2.0.0",
      "configurationUpdate": {
        "merge": "\"mqttTopicMapping\":{\"HelloWorldPubsub\":{\"topic\":\"clients/
MyClientDevice1/hello/world\",\"source\":\"LocalMqtt\",\"target\":\"Pubsub\"}}}"
      }
    }
    ...
  }
}
```

You can use MQTT topic wildcards to relay messages on topics that match a topic filter. If you use MQTT bridge v2.2.0 or later, you can use MQTT topic wildcards in topic filters when the source broker is local publish/subscribe. For more information, see MQTT bridge component configuration.

Receive MQTT messages from client devices

You can subscribe to the local publish/subscribe topics that you configure for the MQTT bridge component to receive messages from client devices.

To receive MQTT messages from client devices in custom components

1. Configure and deploy the MQTT bridge component to relay messages from an MQTT topic where client devices publish to a local publish/subscribe topic.
2. Use the local publish/subscribe IPC interface to subscribe to the topic where the MQTT bridge relays messages. For more information, see Publish/subscribe local messages and SubscribeToTopic.

The Connect and test client devices tutorial includes a section where you develop a component that subscribes to messages from a client device. For more information, see Step 4: Develop a component that communicates with client devices.

Send MQTT messages to client devices

You can publish to the local publish/subscribe topics that you configure for the MQTT bridge component to send messages to client devices.

To publish MQTT messages to client devices in custom components

1. Configure and deploy the MQTT bridge component to relay messages from a local publish/subscribe topic to an MQTT topic where client devices subscribe.
2. Use the local publish/subscribe IPC interface to publish to the topic where the MQTT bridge relays messages. For more information, see Publish/subscribe local messages and PublishToTopic.

Interact with and sync client device shadows

You can use the shadow manager component to manage local shadows, including client device shadows. You can use shadow manager to do the following:

- Interact with client device shadows in Greengrass components.
- Sync client device shadows with AWS IoT Core.

> **Note**
>
> The shadow manager component doesn't sync shadows with AWS IoT Core by default. You must configure the shadow manager component to specify which client device shadows to sync.

Topics

- Prerequisites
- Enable shadow manager to communicate with client devices
- Interact with client device shadows in components
- Sync client device shadows with AWS IoT Core

Prerequisites

To interact with client device shadows and sync client device shadows with AWS IoT Core, a core device must meet the following requirements:

- The core device must run the following components, in addition to the Greengrass components for client device support:
  - Greengrass nucleus v2.6.0 or later
  - Shadow manager v2.2.0 or later
  - MQTT bridge v2.2.0 or later
- The client device auth component must be configured to allow client devices to communicate on device shadow topics.

Enable shadow manager to communicate with client devices

By default, the shadow manager component doesn't manage client device shadows. To enable this feature, you must relay MQTT messages between client devices and the shadow manager component. Client devices use MQTT messages to receive and send device shadow updates. The shadow manager component subscribes to the local Greengrass publish/subscribe interface, so you can configure the MQTT bridge component to relay MQTT messages on device shadow topics.

The MQTT bridge component consumes a list of topic mappings that each specify a message source and a message destination. To enable the shadow manager component to manage client device shadows, deploy the MQTT bridge component, and specify the shadow topics for the client device shadows. You must configure the bridge to relay messages in both directions between local MQTT and local publish/subscribe.

To deploy the MQTT bridge component to a core device or group of core devices, create a deployment that includes the `aws.greengrass.clientdevices.mqtt.Bridge` component. Specify the topic mappings, `mqttTopicMapping`, in the MQTT bridge component configuration in the deployment.

Use the following examples to configure the MQTT bridge component to enable communication between client devices and the shadow manager component.

Note

You can use these configuration examples in the AWS IoT Greengrass console. If you use the AWS IoT Greengrass API, the `merge` configuration update requires a serialized JSON object, so you must serialize the following JSON objects into strings. For more information, see Update component configurations.

Example: Manage all client device shadows

The following MQTT bridge configuration example enables shadow manager to manage all shadows for all client devices.

```
{
  "mqttTopicMapping": {
    "ShadowsLocalMqttToPubsub": {
      "topic": "$aws/things/+/shadow/#",
      "source": "LocalMqtt",
      "target": "Pubsub"
    },
    "ShadowsPubsubToLocalMqtt": {
      "topic": "$aws/things/+/shadow/#",
      "source": "Pubsub",
      "target": "LocalMqtt"
    }
  }
}
```

Example: Manage shadows for a client device

The following MQTT bridge configuration example enables shadow manager to manage all shadows for a client device named `MyClientDevice`.

```
{
  "mqttTopicMapping": {
    "ShadowsLocalMqttToPubsub": {
      "topic": "$aws/things/MyClientDevice/shadow/#",
      "source": "LocalMqtt",
      "target": "Pubsub"
    },
    "ShadowsPubsubToLocalMqtt": {
```

```
      "topic": "$aws/things/MyClientDevice/shadow/#",
      "source": "Pubsub",
      "target": "LocalMqtt"
    }
  }
}
```

Example: Manage a named shadow for all client devices

The following MQTT bridge configuration example enables shadow manager to manage a shadow named `DeviceConfiguration` for all client devices.

```
{
  "mqttTopicMapping": {
    "ShadowsLocalMqttToPubsub": {
      "topic": "$aws/things/+/shadow/name/DeviceConfiguration/#",
      "source": "LocalMqtt",
      "target": "Pubsub"
    },
    "ShadowsPubsubToLocalMqtt": {
      "topic": "$aws/things/+/shadow/name/DeviceConfiguration/#",
      "source": "Pubsub",
      "target": "LocalMqtt"
    }
  }
}
```

Example: Manage all client devices' unnamed shadows

The following MQTT bridge configuration example enables shadow manager to manage unnamed shadows, but not named shadows, for all client devices.

```
{
  "mqttTopicMapping": {
    "DeleteShadowLocalMqttToPubsub": {
      "topic": "$aws/things/+/shadow/delete",
      "source": "LocalMqtt",
      "target": "Pubsub"
    },
    "DeleteShadowPubsubToLocalMqtt": {
      "topic": "$aws/things/+/shadow/delete/#",
      "source": "Pubsub",
      "target": "LocalMqtt"
```

```
    },
    "GetShadowLocalMqttToPubsub": {
      "topic": "$aws/things/+/shadow/get",
      "source": "LocalMqtt",
      "target": "Pubsub"
    },
    "GetShadowPubsubToLocalMqtt": {
      "topic": "$aws/things/+/shadow/get/#",
      "source": "Pubsub",
      "target": "LocalMqtt"
    },
    "UpdateShadowLocalMqttToPubsub": {
      "topic": "$aws/things/+/shadow/update",
      "source": "LocalMqtt",
      "target": "Pubsub"
    },
    "UpdateShadowPubsubToLocalMqtt": {
      "topic": "$aws/things/+/shadow/update/#",
      "source": "Pubsub",
      "target": "LocalMqtt"
    }
  }
}
```

Interact with client device shadows in components

You can develop custom components that use the local shadow service to read and modify client devices' local shadow documents. For more information, see Interact with shadows in components.

Sync client device shadows with AWS IoT Core

You can configure the shadow manager component to synchronize local client device shadow states with AWS IoT Core. For more information, see Sync local device shadows with AWS IoT Core.

Use IPv6 for local messaging

You can configure the IP detector component to use IPv6 to send local messages.

> **Note**
>
> You must have IP detector v2.2.0 or later to use IPv6 to send local messages.

You can deploy the IP detector component to detect and use IPv6 addresses. You must update the configuration of the IP detector component to use IPv6 instead of IPv4. For more information, see Use IP detector to automatically manage endpoints.

Topics

- Configure IP detector to use IPv6

Configure IP detector to use IPv6

If you have a simple network setup, such as the client devices on the same network as the core device, you can deploy the IP detector component to use IPv6 for local messaging.

The IP detector component overwrites endpoints that you set manually.

> **Important**
>
> The core device's AWS IoT policy must allow the `greengrass:UpdateConnectivityInfo` permission to use the IP detector component. For more information, see AWS IoT policies for data plane operations and Configure the AWS IoT thing policy.

You can do either of the following to deploy the IP detector component:

- Use the **Configure discovery** page in the console. For more information, see Configure cloud discovery (console).
- Create and revise deployments to include the IP detector. You can use the console, AWS CLI, or AWS API to manage deployments. For more information, see Create deployments.

Deploy the IP detector component (console)

1. In the AWS IoT Greengrass console navigation menu, choose **Components**.
2. On the **Components** page, choose the **Public components** tab, and then choose **aws.greengrass.clientdevices.IPDetector**.
3. On the **aws.greengrass.clientdevices.IPDetector** page, choose **Deploy**.
4. From **Add to deployment**, choose an existing deployment to revise, or choose to create a new deployment, and then choose **Next**.

5. If you chose to create a new deployment, choose the target core device or thing group for the deployment. On the **Specify target** page, under **Deployment target**, choose a core device or thing group, and then choose **Next**.
6. On the **Select components** page, verify that the **aws.greengrass.clientdevices.IPDetector** component is selected, choose **Next**.
7. On the **Configure components** page, select **aws.greengrass.clientdevices.IPDetector**, and then do the following:

   a. Choose **Configure component**.

   b. In the **Configure aws.greengrass.clientdevices.IPDetector** modal, under **Configuration update**, in **Configuration to merge**, you can enter a configuration update to configure the IP detector component. You can specify any of the following configuration options. Set `includeIPv4Addrs` to `false` and `includeIPv6Addrs` to `true`. You can then update the other IPv6 configuration options.

      - `defaultPort` – (Optional) The MQTT broker port to report when this component detects IP addresses. You must specify this parameter if you configure the MQTT broker to use a different port than the default port 8883.
      - `includeIPv4LoopbackAddrs` – (Optional) You can enable this option to detect and report IPv4 loopback addresses. These are IP addresses, such as `localhost`, where a device can communicate with itself. Use this option in test environments where the core device and client device run on the same system.
      - `includeIPv4LinkLocalAddrs` – (Optional) You can enable this option to detect and report IPv4 link-local addresses. Use this option if the core device's network doesn't have Dynamic Host Configuration Protocol (DHCP) or statically assigned IP addresses.
      - `includeIPv6LoopbackAddrs` – (Optional) You can enable this option to detect and report IPv6 loopback addresses. These are IP addresses, such as `localhost`, where a device can communicate with itself. Use this option in test environments where the core device and client device run on the same system. You must set `includeIPv4Addrs` to `false` and `includeIPv6Addrs` to `true` to use this option. You must have IP detector v2.2.0 or later to use this option.
      - `includeIPv6LinkLocalAddrs` – (Optional) You can enable this option to detect and report IPv6 link-local addresses. Use this option if the core device's network doesn't have Dynamic Host Configuration Protocol (DHCP) or statically assigned IP addresses. You must set `includeIPv4Addrs` to `false` and `includeIPv6Addrs` to `true` to use this option. You must have IP detector v2.2.0 or later to use this option.

- `includeIPv4Addrs` – (Optional) The default is set to true. You can enable this option to publish IPv4 addresses found on the core device. You must have IP detector v2.2.0 or later to use this option.
- `includeIPv6Addrs` – (Optional) You can enable this option to publish IPv6 addresses found on the core device. Set `includeIPv4Addrs` to `false` to use this option. You must have IP detector v2.2.0 or later to use this option.

The configuration update might look similar to the following example.

```
{
  "defaultPort": "8883",
  "includeIPv4LoopbackAddrs": false,
  "includeIPv4LinkLocalAddrs": false,
  "includeIPv6LoopbackAddrs": true,
  "includeIPv6LinkLocalAddrs": true,
  "includeIPv4Addrs": false,
  "includeIPv6Addrs": true
}
```

c. Choose **Confirm** to close the modal, and then choose **Next**.

8. On the **Configure advanced settings** page, keep the default configuration settings, and choose **Next**.
9. On the **Review** page, choose **Deploy**.

   The deployment can take up to a minute to complete.

Deploy the IP detector component (AWS CLI)

To deploy the IP detector component, create a deployment document that includes `aws.greengrass.clientdevices.IPDetector` in the `components` object, and specify the configuration update for the component. Follow instructions in Create deployments to create a new deployment or revise an existing deployment.

You can specify any of the following options to configure the IP detector component when you create the deployment document:

- `defaultPort` – (Optional) The MQTT broker port to report when this component detects IP addresses. You must specify this parameter if you configure the MQTT broker to use a different port than the default port 8883.
- `includeIPv4LoopbackAddrs` – (Optional) You can enable this option to detect and report IPv4 loopback addresses. These are IP addresses, such as `localhost`, where a device can communicate with itself. Use this option in test environments where the core device and client device run on the same system.
- `includeIPv4LinkLocalAddrs` – (Optional) You can enable this option to detect and report IPv4 link-local addresses. Use this option if the core device's network doesn't have Dynamic Host Configuration Protocol (DHCP) or statically assigned IP addresses.
- `includeIPv6LoopbackAddrs` – (Optional) You can enable this option to detect and report IPv6 loopback addresses. These are IP addresses, such as `localhost`, where a device can communicate with itself. Use this option in test environments where the core device and client device run on the same system. You must set `includeIPv4Addrs` to `false` and `includeIPv6Addrs` to `true` to use this option. You must have IP detector v2.2.0 or later to use this option.
- `includeIPv6LinkLocalAddrs` – (Optional) You can enable this option to detect and report IPv6 link-local addresses. Use this option if the core device's network doesn't have Dynamic Host Configuration Protocol (DHCP) or statically assigned IP addresses. You must set `includeIPv4Addrs` to `false` and `includeIPv6Addrs` to `true` to use this option. You must have IP detector v2.2.0 or later to use this option.
- `includeIPv4Addrs` – (Optional) The default is set to true. You can enable this option to publish IPv4 addresses found on the core device. You must have IP detector v2.2.0 or later to use this option.
- `includeIPv6Addrs` – (Optional) You can enable this option to publish IPv6 addresses found on the core device. Set `includeIPv4Addrs` to `false` to use this option. You must have IP detector v2.2.0 or later to use this option.

The following example partial deployment document specifies to use IPv6.

```
{
  ...,
  "components": {
    ...,
    "aws.greengrass.clientdevices.IPDetector": {
      "componentVersion": "2.1.1",
```

```
      "configurationUpdate": {
        "merge": "{\"defaultPort\":\"8883\",}"
      }
    }
  }
}
```

Troubleshooting client devices

Use the troubleshooting information and solutions in this section to help resolve issues with Greengrass client devices and client device components.

Topics

- Greengrass discovery issues
- MQTT connection issues

Greengrass discovery issues

Use the following information to troubleshoot issues with Greengrass discovery. These issues can occur when client devices use the Greengrass discovery API to identify a Greengrass core device to which they can connect.

Topics

- Greengrass discovery issues (HTTP API)
- Greengrass discovery issues (AWS IoT Device SDK v2 for Python)
- Greengrass discovery issues (AWS IoT Device SDK v2 for C++)
- Greengrass discovery issues (AWS IoT Device SDK v2 for JavaScript)
- Greengrass discovery issues (AWS IoT Device SDK v2 for Java)

Greengrass discovery issues (HTTP API)

Use the following information to troubleshoot issues with Greengrass discovery. You might see these errors if you test the discovery API with cURL.

Topics

- curl: (52) Empty reply from server

- HTTP 403: {"message":null,"traceId":"a1b2c3d4-5678-90ab-cdef-11111EXAMPLE"}
- HTTP 404: {"errorMessage":"The thing provided for discovery was not found"}

curl: (52) Empty reply from server

You might see this error if you specify an inactive AWS IoT certificate in the request.

Check that the client device has an attached certificate, and that the certificate is active. For more information, see Attach a thing or policy to a client certificate and Activate or deactivate a client certificate in the *AWS IoT Core Developer Guide*.

HTTP 403: {"message":null,"traceId":"a1b2c3d4-5678-90ab-cdef-11111EXAMPLE"}

You might see this error if the client device doesn't have permission to call `greengrass:Discover` for itself.

Check that the client device's certificate has a policy that allows `greengrass:Discover`. You can't use thing policy variables (`iot:Connection.Thing.*`) in the `Resource` section for this permission. For more information, see Discovery authentication and authorization.

HTTP 404: {"errorMessage":"The thing provided for discovery was not found"}

You might see this error in the following cases:

- The client device isn't associated to any Greengrass core devices or AWS IoT Greengrass V1 groups.
- None of the client device's associated Greengrass core devices or AWS IoT Greengrass V1 groups have an MQTT broker endpoint.
- None of the client device's associated Greengrass core devices run the client device auth component.

Check that the client device is associated to the core device to which you want it to connect. Then, check that the core device runs the client device auth component and has at least one MQTT broker endpoint. For more information, see the following:

- Associate client devices
- Manage core device endpoints
- Configure cloud discovery (console)

Greengrass discovery issues (AWS IoT Device SDK v2 for Python)

Use the following information to troubleshoot issues with Greengrass discovery in the AWS IoT Device SDK v2 for Python.

Topics

- awscrt.exceptions.AwsCrtError: AWS_ERROR_HTTP_CONNECTION_CLOSED: The connection has closed or is closing.
- awsiot.greengrass_discovery.DiscoveryException: ('Error during discover call: response_code=403', 403)
- awsiot.greengrass_discovery.DiscoveryException: ('Error during discover call: response_code=404', 404)

awscrt.exceptions.AwsCrtError: AWS_ERROR_HTTP_CONNECTION_CLOSED: The connection has closed or is closing.

You might see this error if you specify an inactive AWS IoT certificate in the request.

Check that the client device has an attached certificate, and that the certificate is active. For more information, see Attach a thing or policy to a client certificate and Activate or deactivate a client certificate in the *AWS IoT Core Developer Guide*.

awsiot.greengrass_discovery.DiscoveryException: ('Error during discover call: response_code=403', 403)

You might see this error if the client device doesn't have permission to call `greengrass:Discover` for itself.

Check that the client device's certificate has a policy that allows `greengrass:Discover`. You can't use thing policy variables (`iot:Connection.Thing.*`) in the `Resource` section for this permission. For more information, see Discovery authentication and authorization.

awsiot.greengrass_discovery.DiscoveryException: ('Error during discover call: response_code=404', 404)

You might see this error in the following cases:

- The client device isn't associated to any Greengrass core devices or AWS IoT Greengrass V1 groups.

- None of the client device's associated Greengrass core devices or AWS IoT Greengrass V1 groups have an MQTT broker endpoint.
- None of the client device's associated Greengrass core devices run the client device auth component.

Check that the client device is associated to the core device to which you want it to connect. Then, check that the core device runs the client device auth component and has at least one MQTT broker endpoint. For more information, see the following:

- Associate client devices
- Manage core device endpoints
- Configure cloud discovery (console)

Greengrass discovery issues (AWS IoT Device SDK v2 for C++)

Use the following information to troubleshoot issues with Greengrass discovery in the AWS IoT Device SDK v2 for C++.

Topics

- aws-c-http: AWS_ERROR_HTTP_CONNECTION_CLOSED, The connection has closed or is closing.
- aws-c-common: AWS_ERROR_UNKNOWN, Unknown error. (HTTP 403)
- aws-c-common: AWS_ERROR_UNKNOWN, Unknown error. (HTTP 404)

aws-c-http: AWS_ERROR_HTTP_CONNECTION_CLOSED, The connection has closed or is closing.

You might see this error if you specify an inactive AWS IoT certificate in the request.

Check that the client device has an attached certificate, and that the certificate is active. For more information, see Attach a thing or policy to a client certificate and Activate or deactivate a client certificate in the *AWS IoT Core Developer Guide*.

aws-c-common: AWS_ERROR_UNKNOWN, Unknown error. (HTTP 403)

You might see this error if the client device doesn't have permission to call `greengrass:Discover` for itself.

Check that the client device's certificate has a policy that allows `greengrass:Discover`. You can't use thing policy variables (`iot:Connection.Thing.*`) in the `Resource` section for this permission. For more information, see Discovery authentication and authorization.

aws-c-common: AWS_ERROR_UNKNOWN, Unknown error. (HTTP 404)

You might see this error in the following cases:

- The client device isn't associated to any Greengrass core devices or AWS IoT Greengrass V1 groups.
- None of the client device's associated Greengrass core devices or AWS IoT Greengrass V1 groups have an MQTT broker endpoint.
- None of the client device's associated Greengrass core devices run the client device auth component.

Check that the client device is associated to the core device to which you want it to connect. Then, check that the core device runs the client device auth component and has at least one MQTT broker endpoint. For more information, see the following:

- Associate client devices
- Manage core device endpoints
- Configure cloud discovery (console)

Greengrass discovery issues (AWS IoT Device SDK v2 for JavaScript)

Use the following information to troubleshoot issues with Greengrass discovery in the AWS IoT Device SDK v2 for JavaScript.

Topics

- Error: aws-c-http: AWS_ERROR_HTTP_CONNECTION_CLOSED, The connection has closed or is closing.
- Error: Discovery failed (headers: [object Object]) { response_code: 403 }
- Error: Discovery failed (headers: [object Object]) { response_code: 404 }
- Error: Discovery failed (headers: [object Object])

Error: aws-c-http: AWS_ERROR_HTTP_CONNECTION_CLOSED, The connection has closed or is closing.

You might see this error if you specify an inactive AWS IoT certificate in the request.

Check that the client device has an attached certificate, and that the certificate is active. For more information, see Attach a thing or policy to a client certificate and Activate or deactivate a client certificate in the *AWS IoT Core Developer Guide*.

Error: Discovery failed (headers: [object Object]) { response_code: 403 }

You might see this error if the client device doesn't have permission to call `greengrass:Discover` for itself.

Check that the client device's certificate has a policy that allows `greengrass:Discover`. You can't use thing policy variables (`iot:Connection.Thing.*`) in the `Resource` section for this permission. For more information, see Discovery authentication and authorization.

Error: Discovery failed (headers: [object Object]) { response_code: 404 }

You might see this error in the following cases:

- The client device isn't associated to any Greengrass core devices or AWS IoT Greengrass V1 groups.
- None of the client device's associated Greengrass core devices or AWS IoT Greengrass V1 groups have an MQTT broker endpoint.
- None of the client device's associated Greengrass core devices run the client device auth component.

Check that the client device is associated to the core device to which you want it to connect. Then, check that the core device runs the client device auth component and has at least one MQTT broker endpoint. For more information, see the following:

- Associate client devices
- Manage core device endpoints
- Configure cloud discovery (console)

Error: Discovery failed (headers: [object Object])

You might see this error (without an HTTP response code) when you run the Greengrass discovery sample. This error can occur for multiple reasons.

- You might see this error if the client device doesn't have permission to call `greengrass:Discover` for itself.

  Check that the client device's certificate has a policy that allows `greengrass:Discover`. You can't use thing policy variables (`iot:Connection.Thing.*`) in the `Resource` section for this permission. For more information, see Discovery authentication and authorization.
- You might see this error in the following cases:
  - The client device isn't associated to any Greengrass core devices or AWS IoT Greengrass V1 groups.
  - None of the client device's associated Greengrass core devices or AWS IoT Greengrass V1 groups have an MQTT broker endpoint.
  - None of the client device's associated Greengrass core devices run the client device auth component.

  Check that the client device is associated to the core device to which you want it to connect. Then, check that the core device runs the client device auth component and has at least one MQTT broker endpoint. For more information, see the following:
  - Associate client devices
  - Manage core device endpoints
  - Configure cloud discovery (console)

Greengrass discovery issues (AWS IoT Device SDK v2 for Java)

Use the following information to troubleshoot issues with Greengrass discovery in the AWS IoT Device SDK v2 for Java.

Topics

- software.amazon.awssdk.crt.CrtRuntimeException: Error Getting Response Status Code from HttpStream. (aws_last_error: AWS_ERROR_HTTP_DATA_NOT_AVAILABLE(2062), This data is not yet available.)
- java.lang.RuntimeException: Error x-amzn-ErrorType(403)

- java.lang.RuntimeException: Error x-amzn-ErrorType(404)

software.amazon.awssdk.crt.CrtRuntimeException: Error Getting Response Status Code from HttpStream. (aws_last_error: AWS_ERROR_HTTP_DATA_NOT_AVAILABLE(2062), This data is not yet available.)

You might see this error if you specify an inactive AWS IoT certificate in the request.

Check that the client device has an attached certificate, and that the certificate is active. For more information, see Attach a thing or policy to a client certificate and Activate or deactivate a client certificate in the *AWS IoT Core Developer Guide*.

java.lang.RuntimeException: Error x-amzn-ErrorType(403)

You might see this error if the client device doesn't have permission to call `greengrass:Discover` for itself.

Check that the client device's certificate has a policy that allows `greengrass:Discover`. You can't use thing policy variables (`iot:Connection.Thing.*`) in the `Resource` section for this permission. For more information, see Discovery authentication and authorization.

java.lang.RuntimeException: Error x-amzn-ErrorType(404)

You might see this error in the following cases:

- The client device isn't associated to any Greengrass core devices or AWS IoT Greengrass V1 groups.
- None of the client device's associated Greengrass core devices or AWS IoT Greengrass V1 groups have an MQTT broker endpoint.
- None of the client device's associated Greengrass core devices run the client device auth component.

Check that the client device is associated to the core device to which you want it to connect. Then, check that the core device runs the client device auth component and has at least one MQTT broker endpoint. For more information, see the following:

- Associate client devices
- Manage core device endpoints
- Configure cloud discovery (console)

MQTT connection issues

Use the following information to troubleshoot issues with client device MQTT connections. These issues can occur when client devices try to connect to a core device over MQTT.

Topics

- io.moquette.broker.Authorizator: Client does not have read permissions on the topic
- MQTT connection issues (Python)
- MQTT connection issues (C++)
- MQTT connection issues (Java)
- MQTT connection issues (JavaScript)

io.moquette.broker.Authorizator: Client does not have read permissions on the topic

You might see this error in the Greengrass logs when a client device tries to subscribe to an MQTT topic where it doesn't have permission. The error message includes the topic.

Check that the client device auth component' configuration includes the following:

- A device group that matches the client device.
- A client device authorization policy for that device group that grants the `mqtt:subscribe` permission for the topic.

For more information about how to deploy and configure the client device auth component, see the following:

- Configure cloud discovery (console)
- Client device auth
- Create deployments

MQTT connection issues (Python)

Use the following information to troubleshoot issues with client device MQTT connections when you use the AWS IoT Device SDK v2 for Python.

Topics

- AWS_ERROR_MQTT_PROTOCOL_ERROR: Protocol error occurred
- AWS_ERROR_MQTT_UNEXPECTED_HANGUP: Unexpected hangup occurred

AWS_ERROR_MQTT_PROTOCOL_ERROR: Protocol error occurred

You might see this error if the client device auth component doesn't define a client device authorization policy that grants the client device permission to connect.

Check that the client device auth component's configuration includes the following:

- A device group that matches the client device.
- A client device authorization policy for that device group that grants the `mqtt:connect` permission for the client device.

For more information about how to deploy and configure the client device auth component, see the following:

- Configure cloud discovery (console)
- Client device auth
- Create deployments

AWS_ERROR_MQTT_UNEXPECTED_HANGUP: Unexpected hangup occurred

You might see this error if the client device auth component doesn't define a client device authorization policy that grants the client device permission to connect.

Check that the client device auth component's configuration includes the following:

- A device group that matches the client device.
- A client device authorization policy for that device group that grants the `mqtt:connect` permission for the client device.

For more information about how to deploy and configure the client device auth component, see the following:

- Configure cloud discovery (console)

- Client device auth
- Create deployments

MQTT connection issues (C++)

Use the following information to troubleshoot issues with client device MQTT connections when you use the AWS IoT Device SDK v2 for C++.

Topics

- AWS_ERROR_MQTT_PROTOCOL_ERROR: Protocol error occurred
- AWS_ERROR_MQTT_UNEXPECTED_HANGUP: Unexpected hangup occurred

AWS_ERROR_MQTT_PROTOCOL_ERROR: Protocol error occurred

You might see this error if the client device auth component doesn't define a client device authorization policy that grants the client device permission to connect.

Check that the client device auth component's configuration includes the following:

- A device group that matches the client device.
- A client device authorization policy for that device group that grants the `mqtt:connect` permission for the client device.

For more information about how to deploy and configure the client device auth component, see the following:

- Configure cloud discovery (console)
- Client device auth
- Create deployments

AWS_ERROR_MQTT_UNEXPECTED_HANGUP: Unexpected hangup occurred

You might see this error if the client device auth component doesn't define a client device authorization policy that grants the client device permission to connect.

Check that the client device auth component's configuration includes the following:

- A device group that matches the client device.
- A client device authorization policy for that device group that grants the `mqtt:connect` permission for the client device.

For more information about how to deploy and configure the client device auth component, see the following:

- Configure cloud discovery (console)
- Client device auth
- Create deployments

MQTT connection issues (Java)

Use the following information to troubleshoot issues with client device MQTT connections when you use the AWS IoT Device SDK v2 for Java.

Topics

- software.amazon.awssdk.crt.mqtt.MqttException: Protocol error occurred
- AWS_ERROR_MQTT_UNEXPECTED_HANGUP: Unexpected hangup occurred

software.amazon.awssdk.crt.mqtt.MqttException: Protocol error occurred

You might see this error if the client device auth component doesn't define a client device authorization policy that grants the client device permission to connect.

Check that the client device auth component's configuration includes the following:

- A device group that matches the client device.
- A client device authorization policy for that device group that grants the `mqtt:connect` permission for the client device.

For more information about how to deploy and configure the client device auth component, see the following:

- Configure cloud discovery (console)
- Client device auth

- Create deployments

AWS_ERROR_MQTT_UNEXPECTED_HANGUP: Unexpected hangup occurred

You might see this error if the client device auth component doesn't define a client device authorization policy that grants the client device permission to connect.

Check that the client device auth component's configuration includes the following:

- A device group that matches the client device.
- A client device authorization policy for that device group that grants the `mqtt:connect` permission for the client device.

For more information about how to deploy and configure the client device auth component, see the following:

- Configure cloud discovery (console)
- Client device auth
- Create deployments

MQTT connection issues (JavaScript)

Use the following information to troubleshoot issues with client device MQTT connections when you use the AWS IoT Device SDK v2 for JavaScript.

Topics

- AWS_ERROR_MQTT_PROTOCOL_ERROR: Protocol error occurred
- AWS_ERROR_MQTT_UNEXPECTED_HANGUP: Unexpected hangup occurred

AWS_ERROR_MQTT_PROTOCOL_ERROR: Protocol error occurred

You might see this error if the client device auth component doesn't define a client device authorization policy that grants the client device permission to connect.

Check that the client device auth component's configuration includes the following:

- A device group that matches the client device.

- A client device authorization policy for that device group that grants the `mqtt:connect` permission for the client device.

For more information about how to deploy and configure the client device auth component, see the following:

- Configure cloud discovery (console)
- Client device auth
- Create deployments

AWS_ERROR_MQTT_UNEXPECTED_HANGUP: Unexpected hangup occurred

You might see this error if the client device auth component doesn't define a client device authorization policy that grants the client device permission to connect.

Check that the client device auth component's configuration includes the following:

- A device group that matches the client device.
- A client device authorization policy for that device group that grants the `mqtt:connect` permission for the client device.

For more information about how to deploy and configure the client device auth component, see the following:

- Configure cloud discovery (console)
- Client device auth
- Create deployments

Interact with device shadows

Greengrass core devices can interact with AWS IoT device shadows using components. A *shadow* is a JSON document that stores the current or desired state information for an AWS IoT thing. Shadows can make a device's state available to other AWS IoT Greengrass components whether the device is connected to AWS IoT or not. Each AWS IoT device has its own classic, unnamed shadow. You can also create multiple named shadows for each device.

Devices and services can create, update, and delete cloud shadows by using MQTT and the reserved MQTT shadow topics, HTTP using the Device Shadow REST API, and the AWS CLI for AWS IoT.

The shadow manager component enables your Greengrass components to create, update, and delete local shadows by using the local shadow service and the local publish/subscribe shadow topics. The shadow manager also manages the storage of these local shadow documents on your core device, and handles the synchronization of shadow state information with cloud shadows.

You can also use the shadow manager component to manage local shadows for client devices that connect to the core device. To enable shadow manager to manage client device shadows, you configure the MQTT bridge component to relay messages between the local MQTT broker and the local publish/subscribe service. For more information, see Interact with and sync client device shadows.

For more information about AWS IoT device shadow concepts, see AWS IoT Device Shadow service in the *AWS IoT Developer Guide*.

Topics

- Interact with shadows in components
- Sync local device shadows with AWS IoT Core

Interact with shadows in components

You can develop custom components, including Lambda function components, that use the local shadow service to read and modify local shadow documents and client device shadow documents.

Custom components interact with the local shadow service using the AWS IoT Greengrass Core IPC libraries in the AWS IoT Device SDK or the AWS IoT Greengrass Component SDK. The shadow manager component enables the local shadow service on your core device.

To deploy the shadow manager component to a Greengrass core device, create a deployment that includes the `aws.greengrass.ShadowManager` component.

> **Note**
>
> By default, deploying the shadow manager component enables local shadow operations only. To enable AWS IoT Greengrass to sync shadow state information for core device shadows or any shadows for client devices to the corresponding cloud shadow documents in AWS IoT Core, you must create a configuration update for the shadow manager component that includes the `synchronize` parameter. For more information, see Sync local device shadows with AWS IoT Core.

Topics

- Retrieve and modify shadow states
- React to shadow state changes

Retrieve and modify shadow states

The shadow IPC operations retrieve and update state information in local shadow documents. The shadow manager component handles the storage of these shadow documents on your core device.

To modify local shadow states

1. Add authorization policies to the recipe for your custom component to allow the component to receive messages on local shadow topics.

   For example authorization policies, see Local shadow IPC authorization policy examples.

2. Use the shadow IPC operations to retrieve and modify shadow state information. For more information about using shadow IPC operations in component code, see Interact with local shadows.

Note

To enable a core device to interact with client device shadows, you must also configure and deploy the MQTT bridge component. For more information, see Enable shadow manager to communicate with client devices.

React to shadow state changes

Greengrass components use the local publish/subscribe interface to communicate on a core device. To enable a custom component to react to shadow state changes, you can subscribe to the local publish/subscribe topics. This allows the component to receive messages on the local shadow topics, and then act on those messages.

Local shadow topics use the same format as the AWS IoT device shadow MQTT topics. For more information about shadow topics, see Device Shadow MQTT topics in the *AWS IoT Developer Guide*.

To react to local shadow state changes

1. Add access control policies to the recipe for your custom component to allow the component to receive messages on local shadow topics.

   For example authorization policies, see Local shadow IPC authorization policy examples.
2. To initiate a custom action in a component, use `SubscribeToTopic` IPC operations to subscribe to the shadow topics on which you want to receive messages. For more information about using local publish/subscribe IPC operations in component code, see Publish/subscribe local messages.
3. To invoke a Lambda function, use the event source configuration to provide the name of the shadow topic and specify that it's a local publish/subscribe topic. For information about creating Lambda function components, see Run AWS Lambda functions.

Note

To enable a core device to interact with client device shadows, you must also configure and deploy the MQTT bridge component. For more information, see Enable shadow manager to communicate with client devices.

Sync local device shadows with AWS IoT Core

The shadow manager component enables AWS IoT Greengrass to sync local device shadow states with AWS IoT Core. You must modify the configuration of the shadow manager component to include the `synchronization` configuration parameter, and specify the AWS IoT thing names for your devices, and the shadows that you want to sync.

When you configure shadow manager to sync shadows, it syncs all state changes for specified shadows, regardless of whether the changes occur in local shadow documents or in cloud shadow documents.

You can also specify whether the shadow manager component syncs shadows in real time or on a periodic interval. By default, the shadow manager component syncs shadows in real time, so the core device sends and receives shadow updates to and from AWS IoT Core when each update occurs. You can configure periodic intervals to reduce bandwidth usage and charges.

Topics

- Prerequisites
- Configure the shadow manager component
- Sync local shadows
- Shadow merge conflict behavior

Prerequisites

To sync local shadows with AWS IoT Core, you must configure the Greengrass core device's AWS IoT policy to allow the following AWS IoT Core shadow policy actions.

- `iot:GetThingShadow`
- `iot:UpdateThingShadow`
- `iot:DeleteThingShadow`

For more information, see the following:

- AWS IoT Core policy actions in the *AWS IoT Developer Guide*
- Minimal AWS IoT policy for AWS IoT Greengrass V2 core devices
- Update a core device's AWS IoT policy

Configure the shadow manager component

The shadow manager requires a list of shadow name mappings to sync shadow state information in local shadow documents to cloud shadow documents in AWS IoT Core.

To sync shadow states, create a deployment that includes the `aws.greengrass.ShadowManager` component, and specify the shadows that you want to sync in the `synchronize` configuration parameter in the shadow manager configuration in the deployment.

> **Note**
>
> To enable a core device to interact with client device shadows, you must also configure and deploy the MQTT bridge component. For more information, see Enable shadow manager to communicate with client devices.

The following example configuration update instructs the shadow manager component to sync the following shadows with AWS IoT Core:

- The classic shadow for the core device
- The named `MyCoreShadow` for the core device
- The classic shadow for an IoT thing named `MyDevice2`
- The named shadows `MyShadowA` and `MyShadowB` for a IoT thing named `MyDevice1`

This configuration update specifies to sync shadows with AWS IoT Core in real time. If you use shadow manager v2.1.0 or later, you can configure the shadow manager component to sync shadows on a periodic interval. To configure this feature, change the sync strategy to `periodic`, and specify a `delay` in seconds for the interval. For more information, see the strategy configuration parameter of the shadow manager component.

This configuration update specifies to sync shadows in both directions between AWS IoT Core and the core device. If you use shadow manager v2.2.0 or later, you can configure the shadow manager component to sync shadows in only one direction. To configure this feature, change the sync `direction` to `deviceToCloud` or `cloudToDevice`. For more information, see the direction configuration parameter of the shadow manager component.

```
{
```

```
  "strategy": {
    "type": "realTime"
  },
  "synchronize": {
    "coreThing": {
      "classic": true,
      "namedShadows": [
        "MyCoreShadow"
      ]
    },
    "shadowDocuments": [
      {
        "thingName": "MyDevice1",
        "classic": false,
        "namedShadows": [
          "MyShadowA",
          "MyShadowB"
        ]
      },
      {
        "thingName": "MyDevice2",
        "classic": true,
        "namedShadows": [ ]
      }
    ],
    "direction": "betweenDeviceAndCloud"
  }
}
```

Sync local shadows

When the Greengrass core device is connected to the AWS IoT cloud, the shadow manager performs the following tasks for the shadows that you specify in the component configuration. The behavior depends on the shadow sync direction configuration option that you specify. By default, shadow manager uses the `betweenDeviceAndCloud` option to sync shadows in both directions. If you use shadow manager v2.2.0 or later, you can configure the core device to sync shadows in only one direction, which can be `cloudToDevice` or `deviceToCloud`.

- If the shadow sync direction configuration is `betweenDeviceAndCloud` or `cloudToDevice`, shadow manager retrieves the reported state information from the cloud shadow document in AWS IoT Core. Then, it updates locally stored shadow documents to synchronize the device state.

- If the shadow sync direction configuration is `betweenDeviceAndCloud` or `deviceToCloud`, shadow manager publishes the device's current state to the cloud shadow document.

Shadow merge conflict behavior

In some cases, such as when the core device is disconnected from the internet, a shadow might change in the local shadow service and in the AWS IoT cloud before the shadow manager synchronizes the changes. As a result, the desired and reported states differ between the local shadow service and the AWS IoT cloud

When the shadow manager synchronizes the shadow, it merges the changes according to the following behavior:

- If you use a version of shadow manager earlier than v2.2.0, or when you specify the `betweenDeviceAndCloud` shadow sync direction, the following behavior applies:
  - When there's a merge conflict in a shadow's desired state, the shadow manager overwrites the conflicting section of the local shadow document with the value from the AWS IoT cloud.
  - When there's a merge conflict in a shadow's reported state, the shadow manager overwrites the conflicting section of the shadow in the AWS IoT cloud with the value from the local shadow document.
- When you specify the `deviceToCloud` shadow sync direction, the shadow manager overwrites the conflicting section of the shadow in the AWS IoT cloud with the value from the local shadow document.
- When you specify the `cloudToDevice` shadow sync direction, the shadow manager overwrites the conflicting section of the local shadow document with the value from the AWS IoT cloud.

Manage data streams on Greengrass core devices

AWS IoT Greengrass stream manager makes it more efficient and reliable to transfer high-volume IoT data to the AWS Cloud. Stream manager processes data streams on the AWS IoT Greengrass Core before it exports them to the AWS Cloud. Stream manager integrates with common edge scenarios, such as machine learning (ML) inference, where the AWS IoT Greengrass Core device processes and analyzes data before it exports the data to the AWS Cloud or local storage destinations.

Stream manager provides a common interface to simplify custom component development so that you don't need to build custom stream management functionality. Your components can use a standardized mechanism to process high-volume streams and manage local data retention policies. You can define policies for storage type, size, and data retention for each stream to control how stream manager processes and exports data.

Stream manager works in environments with intermittent or limited connectivity. You can define bandwidth use, timeout behavior, and how the AWS IoT Greengrass Core handles stream data when it is connected or disconnected. You can also set priorities to control the order in which the AWS IoT Greengrass Core exports streams to the AWS Cloud. This makes it possible for you to handle critical data sooner than other data.

You can configure stream manager to automatically export data to the AWS Cloud for storage or further processing and analysis. Stream manager supports exports to the following AWS Cloud destinations:

- Streams in Amazon Kinesis Data Streams. You can use Kinesis Data Streams to aggregate high-volume data and load it into a data warehouse or MapReduce cluster. For more information, see What is Amazon Kinesis Data Streams? in the *Amazon Kinesis Data Streams Developer Guide*.
- Asset properties in AWS IoT SiteWise. AWS IoT SiteWise lets you collect, organize, and analyze data from industrial equipment at scale. For more information, see What is AWS IoT SiteWise? in the *AWS IoT SiteWise User Guide*.
- Objects in Amazon Simple Storage Service Amazon S3. You can use Amazon S3 to store and retrieve large amounts of data. For more information, see What is Amazon S3? in the *Amazon Simple Storage Service Developer Guide*.

AWS IoT Analytics discontinued

AWS IoT Analytics was discontinued on December 15, 2025. Stream manager exports to AWS IoT Analytics channels no longer function. If you have streams configured with `IoTAnalyticsConfig` export destinations, remove them to avoid continuous failed export attempts. For more information, see the section called "AWS IoT Analytics channels (discontinued)".

Stream management workflow

Your IoT applications interact with stream manager through the Stream Manager SDK.

In a simple workflow, a component on the AWS IoT Greengrass core consumes IoT data, such as time-series temperature and pressure metrics. The component might filter or compress the data, and then call the Stream Manager SDK to write the data to a stream in stream manager. Stream manager can export the stream to the AWS Cloud automatically based on the policies that you define for the stream. Components can also send data directly to local databases or storage repositories.

Your IoT applications can include multiple custom components that read or write to streams. These components can read and write to streams to filter, aggregate, and analyze data on the AWS IoT Greengrass core device. This makes it possible to respond quickly to local events and extract valuable information before the data transfers from the core to the AWS Cloud or local destinations.

To get started, deploy the stream manager component to your AWS IoT Greengrass core device. In the deployment, configure the stream manager component parameters to define settings that apply to all streams on the Greengrass core device. Use these parameters to control how stream manager stores, processes, and exports streams based on your business needs and environment constraints.

After you configure stream manager, you can create and deploy your IoT applications. These are typically custom components that use `StreamManagerClient` in the Stream Manager SDK to create and interact with streams. When you create a stream, you can define per-stream policies, such as export destinations, priority, and persistence.

Requirements

The following requirements apply for using stream manager:

- Stream manager requires a minimum of 70 MB RAM in addition to the AWS IoT Greengrass Core software. Your total memory requirement depends on your workload.
- AWS IoT Greengrass components must use the Stream Manager SDK to interact with stream manager. The Stream Manager SDK is available in the following languages :
  - Stream Manager SDK for Java (v1.1.0 or later)
  - Stream Manager SDK for Node.js (v1.1.0 or later)
  - Stream Manager SDK for Python (v1.1.0 or later)
- AWS IoT Greengrass components must specify the stream manager component (`aws.greengrass.StreamManager`) as a dependency in their recipe to use stream manager.

  > **Note**
  >
  > If you use stream manager to export data to the cloud, you can't upgrade version 2.0.7 of the stream manager component to a version between v2.0.8 and v2.0.11. If you are deploying stream manager for the first time, we strongly recommend that you deploy the latest version of the stream manager component.

- If you define AWS Cloud export destinations for a stream, you must create your export targets and grant access permissions in the Greengrass device role. Depending on the destination, other requirements might also apply. For more information, see:
  - the section called “AWS IoT Analytics channels (discontinued)”
  - the section called “Amazon Kinesis data streams”
  - the section called “AWS IoT SiteWise asset properties”
  - the section called “Amazon S3 objects”

  You are responsible for maintaining these AWS Cloud resources.

Data security

When you use stream manager, be aware of the following security considerations.

Local data security

AWS IoT Greengrass does not encrypt stream data at rest or in transit between local components on the core device.

- **Data at rest**. Stream data is stored locally in a storage directory. For data security, AWS IoT Greengrass relies on file permissions and full-disk encryption, if enabled. You can use the optional STREAM_MANAGER_STORE_ROOT_DIR parameter to specify the storage directory. If you change this parameter later to use a different storage directory, AWS IoT Greengrass does not delete the previous storage directory or its contents.
- **Data in transit locally**. AWS IoT Greengrass does not encrypt stream data in local transit between data sources, AWS IoT Greengrass components, the Stream Manager SDK, and stream manager.
- **Data in transit to the AWS Cloud**. Data streams exported by stream manager to the AWS Cloud use standard AWS service client encryption with Transport Layer Security (TLS).

Client authentication

Stream manager clients use the Stream Manager SDK to communicate with stream manager. When client authentication is enabled, only Greengrass components can interact with streams in stream manager. When client authentication is disabled, any process running on the Greengrass core device can interact with streams in stream manager. You should disable authentication only if your business case requires it.

You use the STREAM_MANAGER_AUTHENTICATE_CLIENT parameter to set the client authentication mode. You can configure this parameter when you deploy the stream manager component to core devices.

	Enabled	Disabled
Parameter value	`true` (default and recommended)	`false`
Allowed clients	Greengrass components on the core device	Greengrass components on the core device

	Enabled	Disabled
		Other processes running on the Greengrass core device

See also

- the section called “Configure stream manager”
- the section called “Use StreamManagerClient to work with streams”
- the section called “Export configurations for supported cloud destinations”

Configure AWS IoT Greengrass stream manager

On Greengrass core devices, stream manager can store, process, and export IoT device data. Stream manager provides parameters that you use to configure runtime settings. These settings apply to all streams on the Greengrass core device. You can use the AWS IoT Greengrass console or API to configure stream manager settings when you deploy the component. Changes take effect after the deployment completes.

Stream manager parameters

Stream manager provides the following parameters that you can configure when you deploy the component to your core devices. All parameters are optional.

Storage directory

Parameter name: `STREAM_MANAGER_STORE_ROOT_DIR`

The absolute path of the local folder used to store streams. This value must start with a forward slash (for example, `/data`).

You must specify an existing folder, and the system user who runs the stream manager component must have permissions to read and write to this folder. For example, you can run the following commands to create and configure a folder, `/var/greengrass/streams`, which you specify as the stream manager root folder. These commands allow the default system user, `ggc_user`, to read and write to this folder.

```
sudo mkdir /var/greengrass/streams
```

```
sudo chown ggc_user /var/greengrass/streams
sudo chmod 700 /var/greengrass/streams
```

For information about securing stream data, see the section called “Local data security”.

Default: *`/greengrass/v2`*`/work/aws.greengrass.StreamManager`

Server port

Parameter name: `STREAM_MANAGER_SERVER_PORT`

The local port number used to communicate with stream manager. The default is `8088`.

You can specify `0` to use a random available port.

Authenticate client

Parameter name: `STREAM_MANAGER_AUTHENTICATE_CLIENT`

Indicates whether clients must be authenticated to interact with stream manager. All interaction between clients and stream manager is controlled by the Stream Manager SDK. This parameter determines which clients can call the Stream Manager SDK to work with streams. For more information, see the section called “Client authentication”.

Valid values are `true` or `false`. The default is `true` (recommended).

- `true`. Allows only Greengrass components as clients. Components use internal AWS IoT Greengrass Core protocols to authenticate with the Stream Manager SDK.
- `false`. Allows any process that runs on the AWS IoT Greengrass Core to be a client. Do not set the value to `false` unless your business case requires it. For example, use `false` only if non-component processes on the core device must communicate directly with stream manager.

Maximum bandwidth

Parameter name: `STREAM_MANAGER_EXPORTER_MAX_BANDWIDTH`

The average maximum bandwidth (in kilobits per second) that can be used to export data. The default allows unlimited use of available bandwidth.

Thread pool size

Parameter name: `STREAM_MANAGER_EXPORTER_THREAD_POOL_SIZE`

The maximum number of active threads that can be used to export data. The default is `5`.

The optimal size depends on your hardware, stream volume, and planned number of export streams. If your export speed is slow, you can adjust this setting to find the optimal size for your hardware and business case. The CPU and memory of your core device hardware are limiting factors. To start, you might try setting this value equal to the number of processor cores on the device.

Be careful not to set a size that's higher than your hardware can support. Each stream consumes hardware resources, so try to limit the number of export streams on constrained devices.

Kinesis inter-call delay

Parameter name: `STREAM_MANAGER_EXPORTER_KINESIS_INTER_CALL_DELAY_MS`

The delay, in milliseconds, that stream manager waits between consecutive requests when it exports streams to Kinesis Data Streams destinations. The default is `1000` (1 second).

This delay applies only when stream manager cannot determine the shard layout of the target data stream, such as when the Greengrass token exchange role does not include the `kinesis:ListShards` permission. When stream manager can determine the shard layout, it uses shard-aware pacing instead of this delay. To turn off the fallback delay, set this parameter to `0`.

JVM arguments

Parameter name: `JVM_ARGS`

Custom Java Virtual Machine arguments to pass to stream manager at startup. Multiple arguments should be separated by spaces.

Use this parameter only when you must override the default settings used by the JVM. For example, you might need to increase the default heap size if you plan to export a large number of streams.

Logging level

Parameter name: `LOG_LEVEL`

The logging level for the component. Choose from the following log levels, listed here in level order:

- `TRACE`
- `DEBUG`
- `INFO`

- `WARN`
- `ERROR`

Default: `INFO`

Minimum size for multipart upload

Parameter name: `STREAM_MANAGER_EXPORTER_S3_DESTINATION_MULTIPART_UPLOAD_MIN_PART_SIZE_BYTES`

The minimum size (in bytes) of a part in a multipart upload to Amazon S3. Stream manager uses this setting and the size of the input file to determine how to batch data in a multipart PUT request. The default and minimum value is `5242880` bytes (5 MB).

> **Note**
>
> Stream manager uses the stream's `sizeThresholdForMultipartUploadBytes` property to determine whether to export to Amazon S3 as a single or multipart upload. User-defined Greengrass components set this threshold when they create a stream that exports to Amazon S3. The default threshold is 5 MB.

See also

- Manage data streams on Greengrass core devices
- Use StreamManagerClient to work with streams
- Export configurations for supported AWS Cloud destinations

Create custom components that use stream manager

Use stream manager in custom Greengrass components to store, process, and export IoT device data. Use the procedures and examples in this section to create component recipes, artifacts, and applications that work with stream manager. For more information about how to develop and test components, see Create AWS IoT Greengrass components.

Topics

- Define component recipes that use stream manager
- Connect to stream manager in application code

Define component recipes that use stream manager

To use stream manager in a custom component, you must define the `aws.greengrass.StreamManager` component as a dependency. You must also provide the Stream Manager SDK. Complete the following tasks to download and use the Stream Manager SDK in the language of your choice.

Use the Stream Manager SDK for Java

The Stream Manager SDK for Java is available as a JAR file that you can use to compile your component. Then, you can create an application JAR that includes the Stream Manager SDK, define the application JAR as a component artifact, and run the application JAR in the component lifecycle.

To use the Stream Manager SDK for Java

1. Download the Stream Manager SDK for Java JAR file.
2. Do one of the following to create component artifacts from your Java application and the Stream Manager SDK JAR file:
   - Build your application as a JAR file that includes the Stream Manager SDK JAR, and run this JAR file in your component recipe.
   - Define the Stream Manager SDK JAR as a component artifact. Add that artifact to the classpath when you run your application in your component recipe.

   Your component recipe might look like the following example. This component runs a modified version of the StreamManagerS3.java example, where `StreamManagerS3.jar` includes the Stream Manager SDK JAR.

   JSON

   ```
   {
     "RecipeFormatVersion": "2020-01-25",
     "ComponentName": "com.example.StreamManagerS3Java",
     "ComponentVersion": "1.0.0",
     "ComponentDescription": "Uses stream manager to upload a file to an S3
    bucket.",
     "ComponentPublisher": "Amazon",
     "ComponentDependencies": {
       "aws.greengrass.StreamManager": {
         "VersionRequirement": "^2.0.0"
   ```

```
    }
  },
  "Manifests": [
    {
      "Lifecycle": {
        "Run": "java -jar {artifacts:path}/StreamManagerS3.jar"
      },
      "Artifacts": [
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.StreamManagerS3Java/1.0.0/StreamManagerS3.jar"
        }
      ]
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.StreamManagerS3Java
ComponentVersion: 1.0.0
ComponentDescription: Uses stream manager to upload a file to an S3 bucket.
ComponentPublisher: Amazon
ComponentDependencies:
  aws.greengrass.StreamManager:
    VersionRequirement: "^2.0.0"
Manifests:
  - Lifecycle:
      Run: java -jar {artifacts:path}/StreamManagerS3.jar
    Artifacts:
      - URI: s3://amzn-s3-demo-bucket/artifacts/
com.example.StreamManagerS3Java/1.0.0/StreamManagerS3.jar
```

For more information about how to develop and test components, see Create AWS IoT Greengrass components.

Use the Stream Manager SDK for Python

The Stream Manager SDK for Python is available as source code that you can include in your component. Create a ZIP file of the Stream Manager SDK, define the ZIP file as a component artifact, and install the SDK's requirements in the component lifecycle.

To use the Stream Manager SDK for Python

1. Clone or download the aws-greengrass-stream-manager-sdk-python repository.

   ```
   git clone git@github.com:aws-greengrass/aws-greengrass-stream-manager-sdk-python.git
   ```

2. Create a ZIP file that contains the `stream_manager` folder, which contains the source code of the Stream Manager SDK for Python. You can provide this ZIP file as a component artifact that the AWS IoT Greengrass Core software unzips when it installs your component. Do the following:

   a. Open the folder that contains the repository that you cloned or downloaded in the previous step.

      ```
      cd aws-greengrass-stream-manager-sdk-python
      ```

   b. Zip the `stream_manager` folder into a ZIP file named `stream_manager_sdk.zip`.

      Linux or Unix

      ```
      zip -rv stream_manager_sdk.zip stream_manager
      ```

      Windows Command Prompt (CMD)

      ```
      tar -acvf stream_manager_sdk.zip stream_manager
      ```

      PowerShell

      ```
      Compress-Archive stream_manager stream_manager_sdk.zip
      ```

   c. Verify that the `stream_manager_sdk.zip` file contains the `stream_manager` folder and its contents. Run the following command to list the contents of the ZIP file.

Linux or Unix

```
unzip -l stream_manager_sdk.zip
```

Windows Command Prompt (CMD)

```
tar -tf stream_manager_sdk.zip
```

The output should look similar to the following.

```
Archive:  aws-greengrass-stream-manager-sdk-python/stream_manager.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
        0  02-24-2021 20:45   stream_manager/
      913  02-24-2021 20:45   stream_manager/__init__.py
     9719  02-24-2021 20:45   stream_manager/utilinternal.py
     1412  02-24-2021 20:45   stream_manager/exceptions.py
     1004  02-24-2021 20:45   stream_manager/util.py
        0  02-24-2021 20:45   stream_manager/data/
   254463  02-24-2021 20:45   stream_manager/data/__init__.py
    26515  02-24-2021 20:45   stream_manager/streammanagerclient.py
---------                     -------
   294026                     8 files
```

3. Copy the Stream Manager SDK artifacts to your component's artifacts folder. In addition to the Stream Manager SDK ZIP file, your component uses the SDK's `requirements.txt` file to install the dependencies of the Stream Manager SDK. Replace *~/greengrass-components* with the path to the folder that you use for local development.

   Linux or Unix

   ```
   cp {stream_manager_sdk.zip,requirements.txt} ~/greengrass-components/artifacts/
   com.example.StreamManagerS3Python/1.0.0/
   ```

   Windows Command Prompt (CMD)

   ```
   robocopy . %USERPROFILE%\greengrass-components\artifacts
   \com.example.StreamManagerS3Python\1.0.0 stream_manager_sdk.zip
   ```

```
robocopy . %USERPROFILE%\greengrass-components\artifacts
\com.example.StreamManagerS3Python\1.0.0 requirements.txt
```

PowerShell

```
cp .\stream_manager_sdk.zip,.\requirements.txt ~\greengrass-components\artifacts
\com.example.StreamManagerS3Python\1.0.0\
```

4. Create your component recipe. In the recipe, do the following:

   a. Define `stream_manager_sdk.zip` and `requirements.txt` as artifacts.

   b. Define your Python application as an artifact.

   c. In the install lifecycle, install the Stream Manager SDK requirements from `requirements.txt`.

   d. In the run lifecycle, append the Stream Manager SDK to PYTHONPATH, and run your Python application.

   Your component recipe might look like the following example. This component runs the stream_manager_s3.py example.

   JSON

```
{
  "RecipeFormatVersion": "2020-01-25",
  "ComponentName": "com.example.StreamManagerS3Python",
  "ComponentVersion": "1.0.0",
  "ComponentDescription": "Uses stream manager to upload a file to an S3
 bucket.",
  "ComponentPublisher": "Amazon",
  "ComponentDependencies": {
    "aws.greengrass.StreamManager": {
      "VersionRequirement": "^2.0.0"
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
```

```
        "install": "pip3 install --user -r {artifacts:path}/requirements.txt",
        "Run": "export PYTHONPATH=$PYTHONPATH:{artifacts:decompressedPath}/
stream_manager_sdk; python3 {artifacts:path}/stream_manager_s3.py"
      },
      "Artifacts": [
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.StreamManagerS3Python/1.0.0/stream_manager_sdk.zip",
          "Unarchive": "ZIP"
        },
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.StreamManagerS3Python/1.0.0/stream_manager_s3.py"
        },
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.StreamManagerS3Python/1.0.0/requirements.txt"
        }
      ]
    },
    {
      "Platform": {
        "os": "windows"
      },
      "Lifecycle": {
        "install": "pip3 install --user -r {artifacts:path}/requirements.txt",
        "Run": "set \"PYTHONPATH=%PYTHONPATH%;{artifacts:decompressedPath}/
stream_manager_sdk\" & py -3 {artifacts:path}/stream_manager_s3.py"
      },
      "Artifacts": [
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.StreamManagerS3Python/1.0.0/stream_manager_sdk.zip",
          "Unarchive": "ZIP"
        },
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.StreamManagerS3Python/1.0.0/stream_manager_s3.py"
        },
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.StreamManagerS3Python/1.0.0/requirements.txt"
        }
      ]
```

```
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.StreamManagerS3Python
ComponentVersion: 1.0.0
ComponentDescription: Uses stream manager to upload a file to an S3 bucket.
ComponentPublisher: Amazon
ComponentDependencies:
  aws.greengrass.StreamManager:
    VersionRequirement: "^2.0.0"
Manifests:
  - Platform:
      os: linux
    Lifecycle:
      install: pip3 install --user -r {artifacts:path}/requirements.txt
      Run: |
        export PYTHONPATH=$PYTHONPATH:{artifacts:decompressedPath}/
stream_manager_sdk
        python3 {artifacts:path}/stream_manager_s3.py
    Artifacts:
      - URI: s3://amzn-s3-demo-bucket/artifacts/
com.example.StreamManagerS3Python/1.0.0/stream_manager_sdk.zip
        Unarchive: ZIP
      - URI: s3://amzn-s3-demo-bucket/artifacts/
com.example.StreamManagerS3Python/1.0.0/stream_manager_s3.py
      - URI: s3://amzn-s3-demo-bucket/artifacts/
com.example.StreamManagerS3Python/1.0.0/requirements.txt
  - Platform:
      os: windows
    Lifecycle:
      install: pip3 install --user -r {artifacts:path}/requirements.txt
      Run: |
        set "PYTHONPATH=%PYTHONPATH%;{artifacts:decompressedPath}/
stream_manager_sdk"
        py -3 {artifacts:path}/stream_manager_s3.py
    Artifacts:
      - URI: s3://amzn-s3-demo-bucket/artifacts/
com.example.StreamManagerS3Python/1.0.0/stream_manager_sdk.zip
```

```
        Unarchive: ZIP
      - URI: s3://amzn-s3-demo-bucket/artifacts/
com.example.StreamManagerS3Python/1.0.0/stream_manager_s3.py
      - URI: s3://amzn-s3-demo-bucket/artifacts/
com.example.StreamManagerS3Python/1.0.0/requirements.txt
```

For more information about how to develop and test components, see Create AWS IoT Greengrass components.

Use the Stream Manager SDK for JavaScript

The Stream Manager SDK for JavaScript is available as source code that you can include in your component. Create a ZIP file of the Stream Manager SDK, define the ZIP file as a component artifact, and install the SDK in the component lifecycle.

To use the Stream Manager SDK for JavaScript

1. Clone or download the aws-greengrass-stream-manager-sdk-js repository.

   ```
   git clone git@github.com:aws-greengrass/aws-greengrass-stream-manager-sdk-js.git
   ```

2. Create a ZIP file that contains the `aws-greengrass-stream-manager-sdk` folder, which contains the source code of the Stream Manager SDK for JavaScript. You can provide this ZIP file as a component artifact that the AWS IoT Greengrass Core software unzips when it installs your component. Do the following:

   a. Open the folder that contains the repository that you cloned or downloaded in the previous step.

      ```
      cd aws-greengrass-stream-manager-sdk-js
      ```

   b. Zip the `aws-greengrass-stream-manager-sdk` folder into a ZIP file named `stream-manager-sdk.zip`.

      Linux or Unix

      ```
      zip -rv stream-manager-sdk.zip aws-greengrass-stream-manager-sdk
      ```

Windows Command Prompt (CMD)

```
tar -acvf stream-manager-sdk.zip aws-greengrass-stream-manager-sdk
```

PowerShell

```
Compress-Archive aws-greengrass-stream-manager-sdk stream-manager-sdk.zip
```

c. Verify that the `stream-manager-sdk.zip` file contains the `aws-greengrass-stream-manager-sdk` folder and its contents. Run the following command to list the contents of the ZIP file.

Linux or Unix

```
unzip -l stream-manager-sdk.zip
```

Windows Command Prompt (CMD)

```
tar -tf stream-manager-sdk.zip
```

The output should look similar to the following.

```
Archive:  stream-manager-sdk.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
        0  02-24-2021 22:36   aws-greengrass-stream-manager-sdk/
      369  02-24-2021 22:36   aws-greengrass-stream-manager-sdk/package.json
     1017  02-24-2021 22:36   aws-greengrass-stream-manager-sdk/util.js
     8374  02-24-2021 22:36   aws-greengrass-stream-manager-sdk/utilInternal.js
     1937  02-24-2021 22:36   aws-greengrass-stream-manager-sdk/exceptions.js
        0  02-24-2021 22:36   aws-greengrass-stream-manager-sdk/data/
   353343  02-24-2021 22:36   aws-greengrass-stream-manager-sdk/data/index.js
    22599  02-24-2021 22:36   aws-greengrass-stream-manager-sdk/client.js
      216  02-24-2021 22:36   aws-greengrass-stream-manager-sdk/index.js
---------                     -------
   387855                     9 files
```

3. Copy the Stream Manager SDK artifact to your component's artifacts folder. Replace *~/greengrass-components* with the path to the folder that you use for local development.

Linux or Unix

```
cp stream-manager-sdk.zip ~/greengrass-components/artifacts/
com.example.StreamManagerS3JS/1.0.0/
```

Windows Command Prompt (CMD)

```
robocopy . %USERPROFILE%\greengrass-components\artifacts
\com.example.StreamManagerS3JS\1.0.0 stream-manager-sdk.zip
```

PowerShell

```
cp .\stream-manager-sdk.zip ~\greengrass-components\artifacts
\com.example.StreamManagerS3JS\1.0.0\
```

4. Create your component recipe. In the recipe, do the following:

   a. Define `stream-manager-sdk.zip` as an artifact.

   b. Define your JavaScript application as an artifact.

   c. In the install lifecycle, install the Stream Manager SDK from the `stream-manager-sdk.zip` artifact. This `npm install` command creates a `node_modules` folder that contains the Stream Manager SDK and its dependencies.

   d. In the run lifecycle, append the `node_modules` folder to `NODE_PATH`, and run your JavaScript application.

   Your component recipe might look like the following example. This component runs the StreamManagerS3 example.

   JSON

   ```
   {
     "RecipeFormatVersion": "2020-01-25",
     "ComponentName": "com.example.StreamManagerS3JS",
     "ComponentVersion": "1.0.0",
     "ComponentDescription": "Uses stream manager to upload a file to an S3
    bucket.",
     "ComponentPublisher": "Amazon",
     "ComponentDependencies": {
   ```

```
    "aws.greengrass.StreamManager": {
      "VersionRequirement": "^2.0.0"
    }
  },
  "Manifests": [
    {
      "Platform": {
        "os": "linux"
      },
      "Lifecycle": {
        "install": "npm install {artifacts:decompressedPath}/stream-manager-sdk/
aws-greengrass-stream-manager-sdk",
        "Run": "export NODE_PATH=$NODE_PATH:{work:path}/node_modules; node
 {artifacts:path}/index.js"
      },
      "Artifacts": [
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.StreamManagerS3JS/1.0.0/stream-manager-sdk.zip",
          "Unarchive": "ZIP"
        },
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.StreamManagerS3JS/1.0.0/index.js"
        }
      ]
    },
    {
      "Platform": {
        "os": "windows"
      },
      "Lifecycle": {
        "install": "npm install {artifacts:decompressedPath}/stream-manager-sdk/
aws-greengrass-stream-manager-sdk",
        "Run": "set \"NODE_PATH=%NODE_PATH%;{work:path}/node_modules\" & node
 {artifacts:path}/index.js"
      },
      "Artifacts": [
        {
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.StreamManagerS3JS/1.0.0/stream-manager-sdk.zip",
          "Unarchive": "ZIP"
        },
        {
```

```
          "URI": "s3://amzn-s3-demo-bucket/artifacts/
com.example.StreamManagerS3JS/1.0.0/index.js"
        }
      ]
    }
  ]
}
```

YAML

```
---
RecipeFormatVersion: '2020-01-25'
ComponentName: com.example.StreamManagerS3JS
ComponentVersion: 1.0.0
ComponentDescription: Uses stream manager to upload a file to an S3 bucket.
ComponentPublisher: Amazon
ComponentDependencies:
  aws.greengrass.StreamManager:
    VersionRequirement: "^2.0.0"
Manifests:
  - Platform:
      os: linux
    Lifecycle:
      install: npm install {artifacts:decompressedPath}/stream-manager-sdk/aws-
greengrass-stream-manager-sdk
      Run: |
        export NODE_PATH=$NODE_PATH:{work:path}/node_modules
        node {artifacts:path}/index.js
    Artifacts:
      - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.StreamManagerS3JS/1.0.0/stream-manager-sdk.zip
        Unarchive: ZIP
      - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.StreamManagerS3JS/1.0.0/index.js
  - Platform:
      os: windows
    Lifecycle:
      install: npm install {artifacts:decompressedPath}/stream-manager-sdk/aws-
greengrass-stream-manager-sdk
      Run: |
        set "NODE_PATH=%NODE_PATH%;{work:path}/node_modules"
        node {artifacts:path}/index.js
    Artifacts:
```

```
      - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.StreamManagerS3JS/1.0.0/stream-manager-sdk.zip
        Unarchive: ZIP
      - URI: s3://DOC-EXAMPLE-BUCKET/artifacts/
com.example.StreamManagerS3JS/1.0.0/index.js
```

For more information about how to develop and test components, see Create AWS IoT Greengrass components.

Connect to stream manager in application code

To connect to stream manager in your application, create an instance of `StreamManagerClient` from the Stream Manager SDK. This client connects to the stream manager component on its default port 8088, or the port that you specify. For more information about how to use `StreamManagerClient` after you create an instance, see Use StreamManagerClient to work with streams.

Example: Connect to stream manager with default port

Java

```
import com.amazonaws.greengrass.streammanager.client.StreamManagerClient;

public class MyStreamManagerComponent {

    void connectToStreamManagerWithDefaultPort() {
        StreamManagerClient client = StreamManagerClientFactory.standard().build();

        // Use the client.
    }
}
```

Python

```
from stream_manager import (
    StreamManagerClient
)

def connect_to_stream_manager_with_default_port():
    client = StreamManagerClient()
```

```
    # Use the client.
```

JavaScript

```
const {
    StreamManagerClient
} = require('aws-greengrass-stream-manager-sdk');

function connectToStreamManagerWithDefaultPort() {
    const client = new StreamManagerClient();

    // Use the client.
}
```

Example: Connect to stream manager with non-default port

If you configure stream manager with a port other than the default, you must use interprocess communication to retrieve the port from the component configuration.

> **Note**
>
> The `port` configuration parameter contains the value that you specify in `STREAM_MANAGER_SERVER_PORT` when you deploy stream manager.

Java

```
void connectToStreamManagerWithCustomPort() {
    EventStreamRPCConnection eventStreamRpcConnection =
 IPCUtils.getEventStreamRpcConnection();
    GreengrassCoreIPCClient greengrassCoreIPCClient = new
 GreengrassCoreIPCClient(eventStreamRpcConnection);
    List<String> keyPath = new ArrayList<>();
    keyPath.add("port");

    GetConfigurationRequest request = new GetConfigurationRequest();
    request.setComponentName("aws.greengrass.StreamManager");
    request.setKeyPath(keyPath);
    GetConfigurationResponse response =
```

```
            greengrassCoreIPCClient.getConfiguration(request,
 Optional.empty()).getResponse().get();
    String port = response.getValue().get("port").toString();
    System.out.print("Stream Manager is running on port: " + port);

    final StreamManagerClientConfig config = StreamManagerClientConfig.builder()

 .serverInfo(StreamManagerServerInfo.builder().port(Integer.parseInt(port)).build()).build()

    StreamManagerClient client =
 StreamManagerClientFactory.standard().withClientConfig(config).build();

    // Use the client.
}
```

Python

```
import awsiot.greengrasscoreipc
from awsiot.greengrasscoreipc.model import (
    GetConfigurationRequest
)
from stream_manager import (
    StreamManagerClient
)

TIMEOUT = 10

def connect_to_stream_manager_with_custom_port():
    # Use IPC to get the port from the stream manager component configuration.
    ipc_client = awsiot.greengrasscoreipc.connect()
    request = GetConfigurationRequest()
    request.component_name = "aws.greengrass.StreamManager"
    request.key_path = ["port"]
    operation = ipc_client.new_get_configuration()
    operation.activate(request)
    future_response = operation.get_response()
    response = future_response.result(TIMEOUT)
    stream_manager_port = str(response.value["port"])

    # Use port to create a stream manager client.
    stream_client = StreamManagerClient(port=stream_manager_port)

    # Use the client.
```

Use StreamManagerClient to work with streams

User-defined Greengrass components that run on the Greengrass core device can use the `StreamManagerClient` object in the Stream Manager SDK to create streams in stream manager and then interact with the streams. When a component creates a stream, it defines the AWS Cloud destinations, prioritization, and other export and data retention policies for the stream. To send data to stream manager, components append the data to the stream. If an export destination is defined for the stream, stream manager exports the stream automatically.

> **Note**
>
> Typically, clients of stream manager are user-defined Greengrass components. If your business case requires it, you can also allow non-component processes running on the Greengrass core (for example, a Docker container) to interact with stream manager. For more information, see the section called "Client authentication".

The snippets in this topic show you how clients call `StreamManagerClient` methods to work with streams. For implementation details about the methods and their arguments, use the links to the SDK reference listed after each snippet.

If you use stream manager in a Lambda function, your Lambda function should instantiate `StreamManagerClient` outside of the function handler. If instantiated in the handler, the function creates a `client` and connection to stream manager every time that it's invoked.

> **Note**
>
> If you do instantiate `StreamManagerClient` in the handler, you must explicitly call the `close()` method when the `client` completes its work. Otherwise, the `client` keeps the connection open and another thread running until the script exits.

`StreamManagerClient` supports the following operations:

- the section called "Create message stream"
- the section called "Append message"
- the section called "Read messages"

- the section called “List streams”
- the section called “Describe message stream”
- the section called “Update message stream”
- the section called “Delete message stream”

Create message stream

To create a stream, a user-defined Greengrass component calls the create method and passes in a `MessageStreamDefinition` object. This object specifies the unique name for the stream and defines how stream manager should handle new data when the maximum stream size is reached. You can use `MessageStreamDefinition` and its data types (such as `ExportDefinition`, `StrategyOnFull`, and `Persistence`) to define other stream properties. These include:

- The target Kinesis Data Streams, AWS IoT SiteWise, and Amazon S3 destinations for automatic exports. For more information, see the section called “Export configurations for supported cloud destinations”.

  > **AWS IoT Analytics discontinued**
  >
  > AWS IoT Analytics was discontinued on December 15, 2025 and is no longer a supported export destination. If you have streams configured with `IoTAnalyticsConfig` export destinations, remove them to avoid continuous failed export attempts. For more information, see the section called “AWS IoT Analytics channels (discontinued)”.

- Export priority. Stream manager exports higher priority streams before lower priority streams.
- Maximum batch size and batch interval for Kinesis Data Streams and AWS IoT SiteWise destinations. Stream manager exports messages when either condition is met.
- Time-to-live (TTL). The amount of time to guarantee that the stream data is available for processing. You should make sure that the data can be consumed within this time period. This is not a deletion policy. The data might not be deleted immediately after TTL period.
- Stream persistence. Choose to save streams to the file system to persist data across core restarts or save streams in memory.
- Starting sequence number. Specify the sequence number of the message to use as the starting message in the export.

For more information about `MessageStreamDefinition`, see the SDK reference for your target language:

- MessageStreamDefinition in the Java SDK
- MessageStreamDefinition in the Node.js SDK
- MessageStreamDefinition in the Python SDK

> **Note**
>
> `StreamManagerClient` also provides a target destination you can use to export streams to an HTTP server. This target is intended for testing purposes only. It is not stable or supported for use in production environments.

After a stream is created, your Greengrass components can append messages to the stream to send data for export and read messages from the stream for local processing. The number of streams that you create depends on your hardware capabilities and business case. One strategy is to create a stream for each target Kinesis data stream or Amazon S3 destination, though you can define multiple targets for a stream. A stream has a durable lifespan.

Requirements

This operation has the following requirements:

- Minimum Stream Manager SDK version: Python: 1.1.0 | Java: 1.1.0 | Node.js: 1.1.0

Examples

The following snippet creates a stream named `StreamName`. It defines stream properties in the `MessageStreamDefinition` and subordinate data types.

Python

```
client = StreamManagerClient()

try:
    client.create_message_stream(MessageStreamDefinition(
        name="StreamName",    # Required.
        max_size=268435456,    # Default is 256 MB.
```

```
        stream_segment_size=16777216,    # Default is 16 MB.
        time_to_live_millis=None,    # By default, no TTL is enabled.
        strategy_on_full=StrategyOnFull.OverwriteOldestData,    # Required.
        persistence=Persistence.File,    # Default is File.
        flush_on_write=False,    # Default is false.
        export_definition=ExportDefinition(    # Optional. Choose where/how the
 stream is exported to the AWS Cloud.
            kinesis=None,
            iot_analytics=None,
            iot_sitewise=None,
            s3_task_executor=None
        )
    ))
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK reference: create_message_stream | MessageStreamDefinition

Java

```
try (final StreamManagerClient client =
 StreamManagerClientFactory.standard().build()) {
    client.createMessageStream(
            new MessageStreamDefinition()
                    .withName("StreamName") // Required.
                    .withMaxSize(268435456L)    // Default is 256 MB.
                    .withStreamSegmentSize(16777216L)    // Default is 16 MB.
                    .withTimeToLiveMillis(null)    // By default, no TTL is enabled.
                    .withStrategyOnFull(StrategyOnFull.OverwriteOldestData)    //
 Required.
                    .withPersistence(Persistence.File)    // Default is File.
                    .withFlushOnWrite(false)    // Default is false.
                    .withExportDefinition(    // Optional. Choose where/how the
 stream is exported to the AWS Cloud.
                            new ExportDefinition()
                                    .withKinesis(null)
                                    .withIotAnalytics(null)
                                    .withIotSitewise(null)
                                    .withS3(null)
                    )
```

```
    );
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK reference: createMessageStream | MessageStreamDefinition

Node.js

```
const client = new StreamManagerClient();
client.onConnected(async () => {
  try {
    await client.createMessageStream(
      new MessageStreamDefinition()
        .withName("StreamName") // Required.
        .withMaxSize(268435456)  // Default is 256 MB.
        .withStreamSegmentSize(16777216)  // Default is 16 MB.
        .withTimeToLiveMillis(null)  // By default, no TTL is enabled.
        .withStrategyOnFull(StrategyOnFull.OverwriteOldestData)  // Required.
        .withPersistence(Persistence.File)  // Default is File.
        .withFlushOnWrite(false)  // Default is false.
        .withExportDefinition(  // Optional. Choose where/how the stream is exported
 to the AWS Cloud.
          new ExportDefinition()
            .withKinesis(null)
            .withIotAnalytics(null)
            .withIotSiteWise(null)
            .withS3(null)
        )
    );
  } catch (e) {
    // Properly handle errors.
  }
});
client.onError((err) => {
  // Properly handle connection errors.
  // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK reference: createMessageStream | MessageStreamDefinition

For more information about configuring export destinations, see the section called "Export configurations for supported cloud destinations".

Append message

To send data to stream manager for export, your Greengrass components append the data to the target stream. The export destination determines the data type to pass to this method.

Requirements

This operation has the following requirements:

- Minimum Stream Manager SDK version: Python: 1.1.0 | Java: 1.1.0 | Node.js: 1.1.0

Examples

Kinesis Data Streams export destinations

The following snippet appends a message to the stream named `StreamName`. For Kinesis Data Streams destinations, your Greengrass components append a blob of data.

This snippet has the following requirements:

- Minimum Stream Manager SDK version: Python: 1.1.0 | Java: 1.1.0 | Node.js: 1.1.0

Python

```
client = StreamManagerClient()

try:
    sequence_number = client.append_message(stream_name="StreamName",
 data=b'Arbitrary bytes data')
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK reference: append_message

Java

```
try (final StreamManagerClient client =
 StreamManagerClientFactory.standard().build()) {
    long sequenceNumber = client.appendMessage("StreamName", "Arbitrary byte
 array".getBytes());
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK reference: appendMessage

Node.js

```
const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        const sequenceNumber = await client.appendMessage("StreamName",
 Buffer.from("Arbitrary byte array"));
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK reference: appendMessage

AWS IoT SiteWise export destinations

The following snippet appends a message to the stream named `StreamName`. For AWS IoT SiteWise destinations, your Greengrass components append a serialized `PutAssetPropertyValueEntry` object. For more information, see the section called “Exporting to AWS IoT SiteWise”.

Note

When you send data to AWS IoT SiteWise, your data must meet the requirements of the `BatchPutAssetPropertyValue` action. For more information, see BatchPutAssetPropertyValue in the *AWS IoT SiteWise API Reference*.

This snippet has the following requirements:

- Minimum Stream Manager SDK version: Python: 1.1.0 | Java: 1.1.0 | Node.js: 1.1.0

Python

```
client = StreamManagerClient()

try:
    # SiteWise requires unique timestamps in all messages and also needs timestamps
 not earlier
    # than 10 minutes in the past. Add some randomness to time and offset.

    # Note: To create a new asset property data, you should use the classes defined
 in the
    # greengrasssdk.stream_manager module.

    time_in_nanos = TimeInNanos(
        time_in_seconds=calendar.timegm(time.gmtime()) - random.randint(0, 60),
 offset_in_nanos=random.randint(0, 10000)
    )
    variant = Variant(double_value=random.random())
    asset = [AssetPropertyValue(value=variant, quality=Quality.GOOD,
 timestamp=time_in_nanos)]
    putAssetPropertyValueEntry =
 PutAssetPropertyValueEntry(entry_id=str(uuid.uuid4()),
 property_alias="PropertyAlias", property_values=asset)
    sequence_number = client.append_message(stream_name="StreamName",
 Util.validate_and_serialize_to_json_bytes(putAssetPropertyValueEntry))
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
```

```
    # Properly handle errors.
```

Python SDK reference: append_message | PutAssetPropertyValueEntry

Java

```
try (final StreamManagerClient client =
 GreengrassClientBuilder.streamManagerClient().build()) {
    Random rand = new Random();
    // Note: To create a new asset property data, you should use the classes defined
 in the
    // com.amazonaws.greengrass.streammanager.model.sitewise package.
    List<AssetPropertyValue> entries = new ArrayList<>() ;

    // IoTSiteWise requires unique timestamps in all messages and also needs
 timestamps not earlier
    // than 10 minutes in the past. Add some randomness to time and offset.
    final int maxTimeRandomness = 60;
    final int maxOffsetRandomness = 10000;
    double randomValue = rand.nextDouble();
    TimeInNanos timestamp = new TimeInNanos()
            .withTimeInSeconds(Instant.now().getEpochSecond() -
 rand.nextInt(maxTimeRandomness))
            .withOffsetInNanos((long) (rand.nextInt(maxOffsetRandomness)));
    AssetPropertyValue entry = new AssetPropertyValue()
            .withValue(new Variant().withDoubleValue(randomValue))
            .withQuality(Quality.GOOD)
            .withTimestamp(timestamp);
    entries.add(entry);

    PutAssetPropertyValueEntry putAssetPropertyValueEntry = new
 PutAssetPropertyValueEntry()
            .withEntryId(UUID.randomUUID().toString())
            .withPropertyAlias("PropertyAlias")
            .withPropertyValues(entries);
    long sequenceNumber = client.appendMessage("StreamName",
 ValidateAndSerialize.validateAndSerializeToJsonBytes(putAssetPropertyValueEntry));
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK reference: appendMessage | PutAssetPropertyValueEntry

Node.js

```
const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        const maxTimeRandomness = 60;
        const maxOffsetRandomness = 10000;
        const randomValue = Math.random();
        // Note: To create a new asset property data, you should use the classes
 defined in the
        // aws-greengrass-core-sdk StreamManager module.
        const timestamp = new TimeInNanos()
            .withTimeInSeconds(Math.round(Date.now() / 1000) -
 Math.floor(Math.random() - maxTimeRandomness))
            .withOffsetInNanos(Math.floor(Math.random() * maxOffsetRandomness));
        const entry = new AssetPropertyValue()
            .withValue(new Variant().withDoubleValue(randomValue))
            .withQuality(Quality.GOOD)
            .withTimestamp(timestamp);

        const putAssetPropertyValueEntry =    new PutAssetPropertyValueEntry()
            .withEntryId(`${ENTRY_ID_PREFIX}${i}`)
            .withPropertyAlias("PropertyAlias")
            .withPropertyValues([entry]);
        const sequenceNumber = await client.appendMessage("StreamName",
 util.validateAndSerializeToJsonBytes(putAssetPropertyValueEntry));
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK reference: appendMessage | PutAssetPropertyValueEntry

Amazon S3 export destinations

The following snippet appends an export task to the stream named `StreamName`. For Amazon S3 destinations, your Greengrass components append a serialized `S3ExportTaskDefinition` object that contains information about the source input file and target Amazon S3 object. If the

specified object doesn't exist, Stream Manager creates it for you. For more information, see the section called “Exporting to Amazon S3”.

This snippet has the following requirements:

- Minimum Stream Manager SDK version: Python: 1.1.0 | Java: 1.1.0 | Node.js: 1.1.0

Python

```
client = StreamManagerClient()

try:
    # Append an Amazon S3 Task definition and print the sequence number.
    s3_export_task_definition = S3ExportTaskDefinition(input_url="URLToFile",
 bucket="BucketName", key="KeyName")
    sequence_number = client.append_message(stream_name="StreamName",
 Util.validate_and_serialize_to_json_bytes(s3_export_task_definition))
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK reference: append_message | S3ExportTaskDefinition

Java

```
try (final StreamManagerClient client =
 GreengrassClientBuilder.streamManagerClient().build()) {
    // Append an Amazon S3 export task definition and print the sequence number.
    S3ExportTaskDefinition s3ExportTaskDefinition = new S3ExportTaskDefinition()
        .withBucket("BucketName")
        .withKey("KeyName")
        .withInputUrl("URLToFile");
    long sequenceNumber = client.appendMessage("StreamName",
 ValidateAndSerialize.validateAndSerializeToJsonBytes(s3ExportTaskDefinition));
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK reference: appendMessage | S3ExportTaskDefinition

Node.js

```
const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
     // Append an Amazon S3 export task definition and print the sequence number.
     const taskDefinition = new S3ExportTaskDefinition()
        .withBucket("BucketName")
        .withKey("KeyName")
        .withInputUrl("URLToFile");
        const sequenceNumber = await client.appendMessage("StreamName",
 util.validateAndSerializeToJsonBytes(taskDefinition)));
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK reference: appendMessage | S3ExportTaskDefinition

Read messages

Read messages from a stream.

Requirements

This operation has the following requirements:

- Minimum Stream Manager SDK version: Python: 1.1.0 | Java: 1.1.0 | Node.js: 1.1.0

Examples

The following snippet reads messages from the stream named `StreamName`. The read method takes an optional `ReadMessagesOptions` object that specifies the sequence number to start reading from, the minimum and maximum numbers to read, and a timeout for reading messages.

Python

```
client = StreamManagerClient()

try:
    message_list = client.read_messages(
        stream_name="StreamName",
        # By default, if no options are specified, it tries to read one message from
 the beginning of the stream.
        options=ReadMessagesOptions(
            desired_start_sequence_number=100,
            # Try to read from sequence number 100 or greater. By default, this is
 0.
            min_message_count=10,
            # Try to read 10 messages. If 10 messages are not available, then
 NotEnoughMessagesException is raised. By default, this is 1.
            max_message_count=100,    # Accept up to 100 messages. By default this
 is 1.
            read_timeout_millis=5000
            # Try to wait at most 5 seconds for the min_messsage_count to be
 fulfilled. By default, this is 0, which immediately returns the messages or an
 exception.
        )
    )
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK reference: read_messages | ReadMessagesOptions

Java

```
try (final StreamManagerClient client =
 StreamManagerClientFactory.standard().build()) {
    List<Message> messages = client.readMessages("StreamName",
            // By default, if no options are specified, it tries to read one message
 from the beginning of the stream.
            new ReadMessagesOptions()
                    // Try to read from sequence number 100 or greater. By default
 this is 0.
                    .withDesiredStartSequenceNumber(100L)
```

```
                    // Try to read 10 messages. If 10 messages are not available,
 then NotEnoughMessagesException is raised. By default, this is 1.
                    .withMinMessageCount(10L)
                    // Accept up to 100 messages. By default this is 1.
                    .withMaxMessageCount(100L)
                    // Try to wait at most 5 seconds for the min_messsage_count to
 be fulfilled. By default, this is 0, which immediately returns the messages or an
 exception.
                    .withReadTimeoutMillis(Duration.ofSeconds(5L).toMillis())
    );
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK reference: readMessages | ReadMessagesOptions

Node.js

```
const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        const messages = await client.readMessages("StreamName",
            // By default, if no options are specified, it tries to read one message
 from the beginning of the stream.
            new ReadMessagesOptions()
                // Try to read from sequence number 100 or greater. By default this
 is 0.
                .withDesiredStartSequenceNumber(100)
                // Try to read 10 messages. If 10 messages are not available, then
 NotEnoughMessagesException is thrown. By default, this is 1.
                .withMinMessageCount(10)
                // Accept up to 100 messages. By default this is 1.
                .withMaxMessageCount(100)
                // Try to wait at most 5 seconds for the minMessageCount to be
 fulfilled. By default, this is 0, which immediately returns the messages or an
 exception.
                .withReadTimeoutMillis(5 * 1000)
        );
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
    // Properly handle connection errors.
```

```
    // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK reference: readMessages | ReadMessagesOptions

List streams

Get the list of streams in stream manager.

Requirements

This operation has the following requirements:

- Minimum Stream Manager SDK version: Python: 1.1.0 | Java: 1.1.0 | Node.js: 1.1.0

Examples

The following snippet gets a list of the streams (by name) in stream manager.

Python

```
client = StreamManagerClient()

try:
    stream_names = client.list_streams()
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK reference: list_streams

Java

```
try (final StreamManagerClient client =
 StreamManagerClientFactory.standard().build()) {
    List<String> streamNames = client.listStreams();
} catch (StreamManagerException e) {
```

```
    // Properly handle exception.
}
```

Java SDK reference: listStreams

Node.js

```
const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        const streams = await client.listStreams();
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK reference: listStreams

Describe message stream

Get metadata about a stream, including the stream definition, size, and export status.

Requirements

This operation has the following requirements:

- Minimum Stream Manager SDK version: Python: 1.1.0 | Java: 1.1.0 | Node.js: 1.1.0

Examples

The following snippet gets metadata about the stream named `StreamName`, including the stream's definition, size, and exporter statuses.

Python

```
client = StreamManagerClient()
```

```
try:
    stream_description = client.describe_message_stream(stream_name="StreamName")
    if stream_description.export_statuses[0].error_message:
        # The last export of export destination 0 failed with some error
        # Here is the last sequence number that was successfully exported
        stream_description.export_statuses[0].last_exported_sequence_number

    if (stream_description.storage_status.newest_sequence_number >
            stream_description.export_statuses[0].last_exported_sequence_number):
        pass
        # The end of the stream is ahead of the last exported sequence number
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK reference: describe_message_stream

Java

```
try (final StreamManagerClient client =
 StreamManagerClientFactory.standard().build()) {
    MessageStreamInfo description = client.describeMessageStream("StreamName");
    String lastErrorMessage =
 description.getExportStatuses().get(0).getErrorMessage();
    if (lastErrorMessage != null && !lastErrorMessage.equals("")) {
        // The last export of export destination 0 failed with some error.
        // Here is the last sequence number that was successfully exported.
        description.getExportStatuses().get(0).getLastExportedSequenceNumber();
    }

    if (description.getStorageStatus().getNewestSequenceNumber() >
            description.getExportStatuses().get(0).getLastExportedSequenceNumber())
 {
        // The end of the stream is ahead of the last exported sequence number.
    }
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK reference: describeMessageStream

Node.js

```
const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        const description = await client.describeMessageStream("StreamName");
        const lastErrorMessage = description.exportStatuses[0].errorMessage;
        if (lastErrorMessage) {
            // The last export of export destination 0 failed with some error.
            // Here is the last sequence number that was successfully exported.
            description.exportStatuses[0].lastExportedSequenceNumber;
        }

        if (description.storageStatus.newestSequenceNumber >
            description.exportStatuses[0].lastExportedSequenceNumber) {
            // The end of the stream is ahead of the last exported sequence number.
        }
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK reference: describeMessageStream

Update message stream

Update properties of an existing stream. You might want to update a stream if your requirements change after the stream was created. For example:

- Add a new export configuration for an AWS Cloud destination.
- Increase the maximum size of a stream to change how data is exported or retained. For example, the stream size in combination with your strategy on full settings might result in data being deleted or rejected before stream manager can process it.
- Pause and resume exports; for example, if export tasks are long running and you want to ration your upload data.

Your Greengrass components follow this high-level process to update a stream:

1. Get the description of the stream.
2. Update the target properties on the corresponding `MessageStreamDefinition` and subordinate objects.
3. Pass in the updated `MessageStreamDefinition`. Make sure to include the complete object definitions for the updated stream. Undefined properties revert to the default values.

   You can specify the sequence number of the message to use as the starting message in the export.

Requirements

This operation has the following requirements:

- Minimum Stream Manager SDK version: Python: 1.1.0 | Java: 1.1.0 | Node.js: 1.1.0

Examples

The following snippet updates the stream named `StreamName`. It updates multiple properties of a stream that exports to Kinesis Data Streams.

Python

```
client = StreamManagerClient()

try:
    message_stream_info = client.describe_message_stream(STREAM_NAME)
    message_stream_info.definition.max_size=536870912
    message_stream_info.definition.stream_segment_size=33554432
    message_stream_info.definition.time_to_live_millis=3600000
    message_stream_info.definition.strategy_on_full=StrategyOnFull.RejectNewData
    message_stream_info.definition.persistence=Persistence.Memory
    message_stream_info.definition.flush_on_write=False
    message_stream_info.definition.export_definition.kinesis=
        [KinesisConfig(
            # Updating Export definition to add a Kinesis Stream configuration.
            identifier=str(uuid.uuid4()), kinesis_stream_name=str(uuid.uuid4()))]
    client.update_message_stream(message_stream_info.definition)
except StreamManagerException:
```

```
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK reference: updateMessageStream | MessageStreamDefinition

Java

```
try (final StreamManagerClient client =
 GreengrassClientBuilder.streamManagerClient().build()) {
    MessageStreamInfo messageStreamInfo = client.describeMessageStream(STREAM_NAME);
    // Update the message stream with new values.
    client.updateMessageStream(
        messageStreamInfo.getDefinition()
            .withStrategyOnFull(StrategyOnFull.RejectNewData) // Required. Updating
 Strategy on full to reject new data.
            // Max Size update should be greater than initial Max Size defined in
 Create Message Stream request
            .withMaxSize(536870912L) // Update Max Size to 512 MB.
            .withStreamSegmentSize(33554432L) // Update Segment Size to 32 MB.
            .withFlushOnWrite(true) // Update flush on write to true.
            .withPersistence(Persistence.Memory) // Update the persistence to
 Memory.
            .withTimeToLiveMillis(3600000L)    // Update TTL to 1 hour.
            .withExportDefinition(
                // Optional. Choose where/how the stream is exported to the AWS
 Cloud.
                messageStreamInfo.getDefinition().getExportDefinition().
                    // Updating Export definition to add a Kinesis Stream
 configuration.
                    .withKinesis(new ArrayList<KinesisConfig>() {{
                        add(new KinesisConfig()
                            .withIdentifier(EXPORT_IDENTIFIER)
                            .withKinesisStreamName("test"));
                        }})
            );
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK reference: update_message_stream | MessageStreamDefinition

Node.js

```
const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        const messageStreamInfo = await c.describeMessageStream(STREAM_NAME);
        await client.updateMessageStream(
            messageStreamInfo.definition
                // Max Size update should be greater than initial Max Size defined
 in Create Message Stream request
                .withMaxSize(536870912)    // Default is 256 MB. Updating Max Size
 to 512 MB.
                .withStreamSegmentSize(33554432)    // Default is 16 MB. Updating
 Segment Size to 32 MB.
                .withTimeToLiveMillis(3600000)    // By default, no TTL is enabled.
 Update TTL to 1 hour.
                .withStrategyOnFull(StrategyOnFull.RejectNewData)    // Required.
 Updating Strategy on full to reject new data.
                .withPersistence(Persistence.Memory)    // Default is File. Update
 the persistence to Memory
                .withFlushOnWrite(true)    // Default is false. Updating to true.
                .withExportDefinition(
                    // Optional. Choose where/how the stream is exported to the AWS
 Cloud.
                    messageStreamInfo.definition.exportDefinition
                        // Updating Export definition to add a Kinesis Stream
 configuration.
                        .withKinesis([new
 KinesisConfig().withIdentifier(uuidv4()).withKinesisStreamName(uuidv4())])
                )
        );
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK reference: updateMessageStream | MessageStreamDefinition

Constraints for updating streams

The following constraints apply when updating streams. Unless noted in the following list, updates take effect immediately.

- You can't update a stream's persistence. To change this behavior, delete the stream and create a stream that defines the new persistence policy.
- You can update the maximum size of a stream only under the following conditions:
  - The maximum size must be greater or equal to the current size of the stream. To find this information, describe the stream and then check the storage status of the returned `MessageStreamInfo` object.
  - The maximum size must be greater than or equal to the stream's segment size.
- You can update the stream segment size to a value less than the maximum size of the stream. The updated setting applies to new segments.
- Updates to the time to live (TTL) property apply to new append operations. If you decrease this value, stream manager might also delete existing segments that exceed the TTL.
- Updates to the strategy on full property apply to new append operations. If you set the strategy to overwrite the oldest data, stream manager might also overwrite existing segments based on the new setting.
- Updates to the flush on write property apply to new messages.
- Updates to export configurations apply to new exports. The update request must include all export configurations that you want to support. Otherwise, stream manager deletes them.
  - When you update an export configuration, specify the identifier of the target export configuration.
  - To add an export configuration, specify a unique identifier for the new export configuration.
  - To delete an export configuration, omit the export configuration.
- To update the starting sequence number of an export configuration in a stream, you must specify a value that's less than the latest sequence number. To find this information, describe the stream and then check the storage status of the returned `MessageStreamInfo` object.

Delete message stream

Deletes a stream. When you delete a stream, all of the stored data for the stream is deleted from the disk.

Requirements

This operation has the following requirements:

- Minimum Stream Manager SDK version: Python: 1.1.0 | Java: 1.1.0 | Node.js: 1.1.0

Examples

The following snippet deletes the stream named `StreamName`.

Python

```
client = StreamManagerClient()

try:
    client.delete_message_stream(stream_name="StreamName")
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK reference: deleteMessageStream

Java

```
try (final StreamManagerClient client =
 StreamManagerClientFactory.standard().build()) {
    client.deleteMessageStream("StreamName");
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK reference: delete_message_stream

Node.js

```
const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
```

```
        await client.deleteMessageStream("StreamName");
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});
```

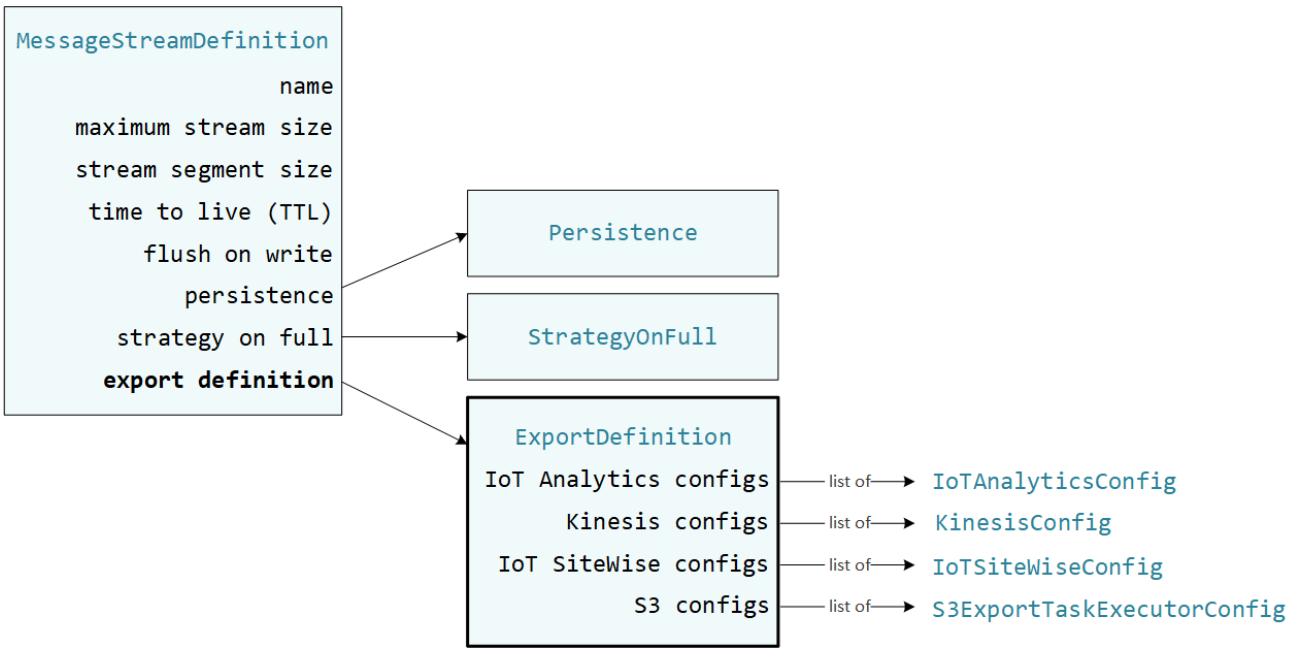
Node.js SDK reference: deleteMessageStream

See also

- Manage data streams on Greengrass core devices
- Configure AWS IoT Greengrass stream manager
- Export configurations for supported AWS Cloud destinations
- `StreamManagerClient` in the Stream Manager SDK reference:
  - Python
  - Java
  - Node.js

Export configurations for supported AWS Cloud destinations

User-defined Greengrass components use `StreamManagerClient` in the Stream Manager SDK to interact with stream manager. When a component creates a stream or updates a stream, it passes a `MessageStreamDefinition` object that represents stream properties, including the export definition. The `ExportDefinition` object contains the export configurations defined for the stream. Stream manager uses these export configurations to determine where and how to export the stream.



You can define zero or more export configurations on a stream, including multiple export configurations for a single destination type. For example, you can export a stream to two Kinesis Data Streams data streams and one Amazon S3 destination.

For failed export attempts, stream manager continually retries exporting data to the AWS Cloud at intervals of up to five minutes. The number of retry attempts doesn't have a maximum limit.

> **Note**
>
> `StreamManagerClient` also provides a target destination you can use to export streams to an HTTP server. This target is intended for testing purposes only. It is not stable or supported for use in production environments.

Supported AWS Cloud destinations

- AWS IoT Analytics channels (discontinued)
- Amazon Kinesis data streams
- AWS IoT SiteWise asset properties
- Amazon S3 objects

You are responsible for maintaining these AWS Cloud resources.

AWS IoT Analytics channels (discontinued)

> **⚠ AWS IoT Analytics discontinued**
>
> AWS IoT Analytics was discontinued on December 15, 2025. Stream manager exports to AWS IoT Analytics channels no longer function. If you have streams configured with `IoTAnalyticsConfig` export destinations, remove them to avoid continuous failed export attempts.

To remove AWS IoT Analytics export configurations from an existing stream, update your Greengrass component to call `updateMessageStream` with an `ExportDefinition` that omits the `IoTAnalyticsConfig` entry. Alternatively, delete and recreate the stream without the AWS IoT Analytics export configuration.

Consider migrating to one of the other supported export destinations, such as Amazon Kinesis Data Streams or Amazon S3, for your data ingestion and analytics workflows.

Amazon Kinesis data streams

Stream manager supports automatic exports to Amazon Kinesis Data Streams. Kinesis Data Streams is commonly used to aggregate high-volume data and load it into a data warehouse or MapReduce cluster. For more information, see What is Amazon Kinesis Data Streams? in the *Amazon Kinesis Developer Guide*.

In the Stream Manager SDK, your Greengrass components use the `KinesisConfig` to define the export configuration for this destination type. For more information, see the SDK reference for your target language:

- KinesisConfig in the Python SDK
- KinesisConfig in the Java SDK
- KinesisConfig in the Node.js SDK

Requirements

This export destination has the following requirements:

- Target streams in Kinesis Data Streams must be in the same AWS account and AWS Region as the Greengrass core device.

- (Recommended) Starting in version 2.2.1, stream manager improves the performance of exporting streams to Kinesis Data Streams destinations. Version 2.3.1 adds adaptive, shard-aware pacing that further increases export throughput. To use these improvements, upgrade your stream manager component to the latest version. Then, grant the `kinesis:ListShards` permission in the Greengrass token exchange role. With this permission, stream manager distributes records across the shards of the target data stream and paces requests to stay within the per-shard limits of Kinesis Data Streams.
- The Authorize core devices to interact with AWS services must allow the `kinesis:PutRecords` permission to target data streams. For example:

  JSON

  ```
  {
    "Version":"2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Action": [
          "kinesis:PutRecords"
        ],
        "Resource": [
          "arn:aws:kinesis:us-east-1:123456789012:stream/stream_1_name",
          "arn:aws:kinesis:us-east-1:123456789012:stream/stream_2_name"
        ]
      }
    ]
  }
  ```

  You can grant granular or conditional access to resources, for example, by using a wildcard * naming scheme. For more information, see Adding and removing IAM policies in the *IAM User Guide*.

Exporting to Kinesis Data Streams

To create a stream that exports to Kinesis Data Streams, your Greengrass components create a stream with an export definition that includes one or more `KinesisConfig` objects. This object defines export settings, such as the target data stream, batch size, batch interval, and priority.

When your Greengrass components receive data from devices, they append messages that contain a blob of data to the target stream. Then, stream manager exports the data based on the batch settings and priority defined in the stream's export configurations.

When stream manager can determine the shard layout of the target data stream, it distributes records across the available shards. Stream manager also paces requests to stay within the per-shard limits of Kinesis Data Streams. If stream manager cannot determine the shard layout, it assigns a unique, random UUID as the partition key for each record that it uploads to Amazon Kinesis.

AWS IoT SiteWise asset properties

Stream manager supports automatic exports to AWS IoT SiteWise. AWS IoT SiteWise lets you collect, organize, and analyze data from industrial equipment at scale. For more information, see What is AWS IoT SiteWise? in the *AWS IoT SiteWise User Guide*.

In the Stream Manager SDK, your Greengrass components use the `IoTSiteWiseConfig` to define the export configuration for this destination type. For more information, see the SDK reference for your target language:

- IoTSiteWiseConfig in the Python SDK
- IoTSiteWiseConfig in the Java SDK
- IoTSiteWiseConfig in the Node.js SDK

> **Note**
>
> AWS also provides AWS IoT SiteWise components, which offer a pre-built solution that you can use to stream data from OPC-UA sources. For more information, see IoT SiteWise OPC UA collector.

Requirements

This export destination has the following requirements:

- Target asset properties in AWS IoT SiteWise must be in the same AWS account and AWS Region as the Greengrass core device.

> **Note**
>
> For the list of AWS Regions that AWS IoT SiteWise supports, see AWS IoT SiteWise endpoints and quotas in the *AWS General Reference*.

- The Authorize core devices to interact with AWS services must allow the `iotsitewise:BatchPutAssetPropertyValue` permission to target asset properties. The following example policy uses the `iotsitewise:assetHierarchyPath` condition key to grant access to a target root asset and its children. You can remove the `Condition` from the policy to allow access to all of your AWS IoT SiteWise assets or specify ARNs of individual assets.

  JSON

  ```
  {
    "Version":"2012-10-17",
    "Statement": [
      {
         "Effect": "Allow",
         "Action": "iotsitewise:BatchPutAssetPropertyValue",
         "Resource": "*",
         "Condition": {
           "StringLike": {
             "iotsitewise:assetHierarchyPath": [
               "/root node asset ID",
               "/root node asset ID/*"
             ]
           }
         }
      }
    ]
  }
  ```

  You can grant granular or conditional access to resources, for example, by using a wildcard * naming scheme. For more information, see Adding and removing IAM policies in the *IAM User Guide*.

  For important security information, see BatchPutAssetPropertyValue authorization in the *AWS IoT SiteWise User Guide*.

Exporting to AWS IoT SiteWise

To create a stream that exports to AWS IoT SiteWise, your Greengrass components create a stream with an export definition that includes one or more `IoTSiteWiseConfig` objects. This object defines export settings, such as the batch size, batch interval, and priority.

When your Greengrass components receive asset property data from devices, they append messages that contain the data to the target stream. Messages are JSON-serialized `PutAssetPropertyValueEntry` objects that contain property values for one or more asset properties. For more information, see Append message for AWS IoT SiteWise export destinations.

> **Note**
>
> When you send data to AWS IoT SiteWise, your data must meet the requirements of the `BatchPutAssetPropertyValue` action. For more information, see BatchPutAssetPropertyValue in the *AWS IoT SiteWise API Reference*.

Then, stream manager exports the data based on the batch settings and priority defined in the stream's export configurations.

You can adjust your stream manager settings and Greengrass component logic to design your export strategy. For example:

- For near real time exports, set low batch size and interval settings and append the data to the stream when it's received.
- To optimize batching, mitigate bandwidth constraints, or minimize cost, your Greengrass components can pool the timestamp-quality-value (TQV) data points received for a single asset property before appending the data to the stream. One strategy is to batch entries for up to 10 different property-asset combinations, or property aliases, in one message instead of sending more than one entry for the same property. This helps stream manager to remain within AWS IoT SiteWise quotas.

Amazon S3 objects

Stream manager supports automatic exports to Amazon S3. You can use Amazon S3 to store and retrieve large amounts of data. For more information, see What is Amazon S3? in the *Amazon Simple Storage Service Developer Guide*.

In the Stream Manager SDK, your Greengrass components use the `S3ExportTaskExecutorConfig` to define the export configuration for this destination type. For more information, see the SDK reference for your target language:

- S3ExportTaskExecutorConfig in the Python SDK
- S3ExportTaskExecutorConfig in the Java SDK
- S3ExportTaskExecutorConfig in the Node.js SDK

Requirements

This export destination has the following requirements:

- Target Amazon S3 buckets must be in the same AWS account as the Greengrass core device.
- If a Lambda function that runs in **Greengrass container** mode writes input files to an input file directory, you must mount the directory as a volume in the container with write permissions. This ensures that the files are written to the root file system and visible to the stream manager component, which runs outside the container.
- If a Docker container component writes input files to an input file directory, you must mount the directory as a volume in the container with write permissions. This ensures that the files are written to the root file system and visible to the stream manager component, which runs outside the container.
- The Authorize core devices to interact with AWS services must allow the following permissions to the target buckets. For example:

  JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:PutObject",
        "s3:AbortMultipartUpload",
        "s3:ListMultipartUploadParts"
      ],
      "Resource": [
        "arn:aws:s3:::bucket-1-name/*",
        "arn:aws:s3:::bucket-2-name/*"
```

```
      ]
    }
  ]
}
```

You can grant granular or conditional access to resources, for example, by using a wildcard * naming scheme. For more information, see Adding and removing IAM policies in the *IAM User Guide*.

Exporting to Amazon S3

To create a stream that exports to Amazon S3, your Greengrass components use the `S3ExportTaskExecutorConfig` object to configure the export policy. The policy defines export settings, such as the multipart upload threshold and priority. For Amazon S3 exports, stream manager uploads data that it reads from local files on the core device. To initiate an upload, your Greengrass components append an export task to the target stream. The export task contains information about the input file and target Amazon S3 object. Stream manager runs tasks in the sequence that they are appended to the stream.

> **Note**
>
> The target bucket must already exist in your AWS account. If an object for the specified key doesn't exist, stream manager creates the object for you.

Stream manager uses the multipart upload threshold property, minimum part size setting, and size of the input file to determine how to upload data. The multipart upload threshold must be greater or equal to the minimum part size. If you want to upload data in parallel, you can create multiple streams.

The keys that specify your target Amazon S3 objects can include valid Java DateTimeFormatter strings in `!{timestamp:`*`value`*`}` placeholders. You can use these timestamp placeholders to partition data in Amazon S3 based on the time that the input file data was uploaded. For example, the following key name resolves to a value such as `my-key/2020/12/31/data.txt`.

```
my-key/!{timestamp:YYYY}/!{timestamp:MM}/!{timestamp:dd}/data.txt
```

> **Note**
>
> If you want to monitor the export status for a stream, first create a status stream and then configure the export stream to use it. For more information, see the section called "Monitor export tasks".

Manage input data

You can author code that IoT applications use to manage the lifecycle of the input data. The following example workflow shows how you might use Greengrass components to manage this data.

1. A local process receives data from devices or peripherals, and then writes the data to files in a directory on the core device. These are the input files for stream manager.
2. A Greengrass component scans the directory and appends an export task to the target stream when a new file is created. The task is a JSON-serialized `S3ExportTaskDefinition` object that specifies the URL of the input file, the target Amazon S3 bucket and key, and optional user metadata.
3. Stream manager reads the input file and exports the data to Amazon S3 in the order of appended tasks. The target bucket must already exist in your AWS account. If an object for the specified key doesn't exist, stream manager creates the object for you.
4. The Greengrass component reads messages from a status stream to monitor the export status. After export tasks are completed, the Greengrass component can delete the corresponding input files. For more information, see the section called "Monitor export tasks".

Monitor export tasks

You can author code that IoT applications use to monitor the status of your Amazon S3 exports. Your Greengrass components must create a status stream and then configure the export stream to write status updates to the status stream. A single status stream can receive status updates from multiple streams that export to Amazon S3.

First, create a stream to use as the status stream. You can configure the size and retention policies for the stream to control the lifespan of the status messages. For example:

- Set `Persistence` to `Memory` if you don't want to store the status messages.

- Set `StrategyOnFull` to `OverwriteOldestData` so that new status messages are not lost.

Then, create or update the export stream to use the status stream. Specifically, set the status configuration property of the stream's `S3ExportTaskExecutorConfig` export configuration. This setting tells stream manager to write status messages about the export tasks to the status stream. In the `StatusConfig` object, specify the name of the status stream and the level of verbosity. The following supported values range from least verbose (`ERROR`) to most verbose (`TRACE`). The default is `INFO`.

- `ERROR`
- `WARN`
- `INFO`
- `DEBUG`
- `TRACE`

The following example workflow shows how Greengrass components might use a status stream to monitor export status.

1. As described in the previous workflow, a Greengrass component appends an export task to a stream that's configured to write status messages about export tasks to a status stream. The append operation return a sequence number that represents the task ID.
2. A Greengrass component reads messages sequentially from the status stream, and then filters the messages based on the stream name and task ID or based on an export task property from the message context. For example, the Greengrass component can filter by the input file URL of the export task, which is represented by the `S3ExportTaskDefinition` object in the message context.

   The following status codes indicate that an export task has reached a completed state:

   - `Success`. The upload was completed successfully.
   - `Failure`. Stream manager encountered an error, for example, the specified bucket does not exist. After resolving the issue, you can append the export task to the stream again.
   - `Canceled`. The task was stopped because the stream or export definition was deleted, or the time-to-live (TTL) period of the task expired.

> **Note**
>
> The task might also have a status of `InProgress` or `Warning`. Stream manager issues warnings when an event returns an error that doesn't affect the execution of the task. For example, a failure to clean up a partial upload returns a warning.

3. After export tasks are completed, the Greengrass component can delete the corresponding input files.

The following example shows how a Greengrass component might read and process status messages.

Python

```
import time
from stream_manager import (
    ReadMessagesOptions,
    Status,
    StatusConfig,
    StatusLevel,
    StatusMessage,
    StreamManagerClient,
)
from stream_manager.util import Util

client = StreamManagerClient()

try:
    # Read the statuses from the export status stream
    is_file_uploaded_to_s3 = False
    while not is_file_uploaded_to_s3:
        try:
            messages_list = client.read_messages(
                "StatusStreamName", ReadMessagesOptions(min_message_count=1,
 read_timeout_millis=1000)
            )
            for message in messages_list:
                # Deserialize the status message first.
                status_message = Util.deserialize_json_bytes_to_obj(message.payload,
 StatusMessage)
```

```
                # Check the status of the status message. If the status is
 "Success",
                # the file was successfully uploaded to S3.
                # If the status was either "Failure" or "Cancelled", the server was
 unable to upload the file to S3.
                # We will print the message for why the upload to S3 failed from the
 status message.
                # If the status was "InProgress", the status indicates that the
 server has started uploading
                # the S3 task.
                if status_message.status == Status.Success:
                    logger.info("Successfully uploaded file at path " + file_url + "
 to S3.")
                    is_file_uploaded_to_s3 = True
                elif status_message.status == Status.Failure or
 status_message.status == Status.Canceled:
                    logger.info(
                        "Unable to upload file at path " + file_url + " to S3.
 Message: " + status_message.message
                    )
                    is_file_uploaded_to_s3 = True
            time.sleep(5)
        except StreamManagerException:
            logger.exception("Exception while running")
except StreamManagerException:
    pass
    # Properly handle errors.
except ConnectionError or asyncio.TimeoutError:
    pass
    # Properly handle errors.
```

Python SDK reference: read_messages | StatusMessage

Java

```
import com.amazonaws.greengrass.streammanager.client.StreamManagerClient;
import com.amazonaws.greengrass.streammanager.client.StreamManagerClientFactory;
import com.amazonaws.greengrass.streammanager.client.utils.ValidateAndSerialize;
import com.amazonaws.greengrass.streammanager.model.ReadMessagesOptions;
import com.amazonaws.greengrass.streammanager.model.Status;
import com.amazonaws.greengrass.streammanager.model.StatusConfig;
import com.amazonaws.greengrass.streammanager.model.StatusLevel;
import com.amazonaws.greengrass.streammanager.model.StatusMessage;
```

```
try (final StreamManagerClient client =
StreamManagerClientFactory.standard().build()) {
   try {
       boolean isS3UploadComplete = false;
       while (!isS3UploadComplete) {
           try {
               // Read the statuses from the export status stream
               List<Message> messages = client.readMessages("StatusStreamName",
                   new
ReadMessagesOptions().withMinMessageCount(1L).withReadTimeoutMillis(1000L));
               for (Message message : messages) {
                   // Deserialize the status message first.
                   StatusMessage statusMessage =
ValidateAndSerialize.deserializeJsonBytesToObj(message.getPayload(),
StatusMessage.class);
                   // Check the status of the status message. If the status is
"Success", the file was successfully uploaded to S3.
                   // If the status was either "Failure" or "Canceled", the server
was unable to upload the file to S3.
                   // We will print the message for why the upload to S3 failed
from the status message.
                   // If the status was "InProgress", the status indicates that the
server has started uploading the S3 task.
                   if (Status.Success.equals(statusMessage.getStatus())) {
                       System.out.println("Successfully uploaded file at path " +
FILE_URL + " to S3.");
                       isS3UploadComplete = true;
                    } else if (Status.Failure.equals(statusMessage.getStatus()) ||
Status.Canceled.equals(statusMessage.getStatus())) {
                       System.out.println(String.format("Unable to upload file at
path %s to S3. Message %s",

statusMessage.getStatusContext().getS3ExportTaskDefinition().getInputUrl(),
                           statusMessage.getMessage()));
                       sS3UploadComplete = true;
                   }
               }
           } catch (StreamManagerException ignored) {
           } finally {
               // Sleep for sometime for the S3 upload task to complete before
trying to read the status message.
               Thread.sleep(5000);
           }
       } catch (e) {
```

```
        // Properly handle errors.
    }
} catch (StreamManagerException e) {
    // Properly handle exception.
}
```

Java SDK reference: readMessages | StatusMessage

Node.js

```
const {
    StreamManagerClient, ReadMessagesOptions,
    Status, StatusConfig, StatusLevel, StatusMessage,
    util,
} = require(*'aws-greengrass-stream-manager-sdk'*);

const client = new StreamManagerClient();
client.onConnected(async () => {
    try {
        let isS3UploadComplete = false;
        while (!isS3UploadComplete) {
            try {
                // Read the statuses from the export status stream
                const messages = await c.readMessages("StatusStreamName",
                    new ReadMessagesOptions()
                        .withMinMessageCount(1)
                        .withReadTimeoutMillis(1000));

                messages.forEach((message) => {
                    // Deserialize the status message first.
                    const statusMessage =
 util.deserializeJsonBytesToObj(message.payload, StatusMessage);
                    // Check the status of the status message. If the status is
 'Success', the file was successfully uploaded to S3.
                    // If the status was either 'Failure' or 'Cancelled', the server
 was unable to upload the file to S3.
                    // We will print the message for why the upload to S3 failed
 from the status message.
                    // If the status was "InProgress", the status indicates that the
 server has started uploading the S3 task.
                    if (statusMessage.status === Status.Success) {
                        console.log(`Successfully uploaded file at path ${FILE_URL}
 to S3.`);
                        isS3UploadComplete = true;
```

```
                    } else if (statusMessage.status === Status.Failure ||
 statusMessage.status === Status.Canceled) {
                        console.log(`Unable to upload file at path ${FILE_URL} to
 S3. Message: ${statusMessage.message}`);
                        isS3UploadComplete = true;
                    }
                });
                // Sleep for sometime for the S3 upload task to complete before
 trying to read the status message.
                await new Promise((r) => setTimeout(r, 5000));
            } catch (e) {
                // Ignored
            }
    } catch (e) {
        // Properly handle errors.
    }
});
client.onError((err) => {
    // Properly handle connection errors.
    // This is called only when the connection to the StreamManager server fails.
});
```

Node.js SDK reference: readMessages | StatusMessage

Perform machine learning inference

With AWS IoT Greengrass, you can perform machine learning (ML) inference on your edge devices on locally generated data using cloud-trained models. You benefit from the low latency and cost savings of running local inference, yet still take advantage of cloud computing power for training models and complex processing.

AWS IoT Greengrass makes the steps required to perform inference more efficient. You can train your inference models anywhere and deploy them locally as *machine learning components*. For example, you can build and train deep-learning models in Amazon SageMaker AI. Then, you can store these models in an Amazon S3 bucket, so you can use these models as artifacts in your components to perform inference on your core devices.

Topics

- How AWS IoT Greengrass ML inference works
- What's different in AWS IoT Greengrass Version 2?
- Requirements
- Supported model sources
- Supported machine learning runtimes
- AWS-provided machine learning components
- Use Amazon SageMaker AI Edge Manager on Greengrass core devices
- Customize your machine learning components
- Troubleshooting machine learning inference

How AWS IoT Greengrass ML inference works

AWS provides machine learning components that you can use to create one-step deployments to perform machine learning inference on your device. You can also use these components as templates to create custom components to meet your specific requirements.

AWS provides the following categories of machine learning components:

- **Model component**—Contains machine learning models as Greengrass artifacts.
- **Runtime component**—Contains the script that installs the machine learning framework and its dependencies on the Greengrass core device.

- **Inference component**—Contains the inference code and includes component dependencies to install the machine learning framework and download pre-trained machine learning models.

Each deployment that you create to perform machine learning inference consists of at least one component that runs your inference application, installs the machine learning framework, and downloads your machine learning models. To perform sample inference with AWS-provided components, you deploy an inference component to your core device, which automatically includes the corresponding model and runtime components as dependencies. To customize your deployments, you can plug in or swap out the sample model components with custom model components, or you can use the component recipes for the AWS-provided components as templates to create your own custom inference, model, and runtime components.

To perform machine learning inference by using custom components:

1. Create a model component. This component contains the machine learning models that you want to use to perform inference. AWS provides sample pre-trained DLR and TensorFlow Lite models. To use a custom model, create your own model component.
2. Create a runtime component. This component contains the scripts required to install the machine learning runtime for your models. AWS provides sample runtime components for Deep Learning Runtime (DLR) and TensorFlow Lite. To use other runtimes with your custom models and inference code, create your own runtime components.
3. Create an inference component. This component contains your inference code, and includes your model and runtime components as dependencies. AWS provides sample inference components for image classification and object detection using DLR and TensorFlow Lite. To perform other types of inference, or to use custom models and runtimes, create your own inference component.
4. Deploy the inference component. When you deploy this component, AWS IoT Greengrass also automatically deploys the model and runtime component dependencies.

To get started with AWS-provided components, see the section called “Perform sample image classification inference”.

For information about creating custom machine learning components, see Customize your machine learning components.

What's different in AWS IoT Greengrass Version 2?

AWS IoT Greengrass consolidates functional units for machine learning—such as models, runtimes, and inference code— into components that enable you to use a one-step process to install the machine learning runtime, download your trained models, and perform inference on your device.

By using the AWS-provided machine learning components, you have the flexibility to start performing machine learning inference with sample inference code and pre-trained models. You can plug in custom model components to use your own custom-trained models with the inference and runtime components that AWS provides. For a completely customized machine learning solution, you can use the public components as templates to create custom components and use any runtime, model, or inference type that you want.

Requirements

To create and use machine learning components, you must have the following:

- A Greengrass core device. If you don't have one, see Tutorial: Getting started with AWS IoT Greengrass V2.
- Minimum 500 MB local storage space to use AWS-provided sample machine learning components.

Supported model sources

AWS IoT Greengrass supports using custom-trained machine learning models that are stored in Amazon S3. You can also use Amazon SageMaker AI edge packaging jobs to directly create model components for your SageMaker AI Neo-compiled models. For information about using SageMaker AI Edge Manager with AWS IoT Greengrass, see Use Amazon SageMaker AI Edge Manager on Greengrass core devices.

The S3 buckets that contain your models must meet the following requirements:

- They must not be encrypted using SSE-C. For buckets that use server-side encryption, AWS IoT Greengrass machine learning inference currently supports the SSE-S3 or SSE-KMS encryption options only. For more information about server-side encryption options, see Protecting data using server-side encryption in the *Amazon Simple Storage Service User Guide*.

- Their names must not include periods (`.`). For more information, see the rule about using virtual hosted-style buckets with SSL in Rules for bucket naming in the *Amazon Simple Storage Service User Guide*.
- The S3 buckets that store your model sources must be in the same AWS account and AWS Region as your machine learning components.
- AWS IoT Greengrass must have `read` permission to the model source. To enable AWS IoT Greengrass to access the S3 buckets, the Greengrass device role must allow the `s3:GetObject` action. For more information about the device role, see Authorize core devices to interact with AWS services.

Supported machine learning runtimes

AWS IoT Greengrass enables you to create custom components to use any machine learning runtime of your choice to perform machine learning inference with your custom-trained models. For information about creating custom machine learning components, see Customize your machine learning components.

To make the process of getting started with machine learning more efficient, AWS IoT Greengrass provides sample inference, model, and runtime components that use the following machine learning runtimes:

- Deep Learning Runtime (DLR) v1.6.0 and v1.3.0
- TensorFlow Lite v2.5.0

AWS-provided machine learning components

The following table lists the AWS-provided components used for machine learning.

> **Note**
>
> Several AWS-provided components depend on specific minor versions of the Greengrass nucleus. Because of this dependency, you need to update these components when you update the Greengrass nucleus to a new minor version. For information about the specific versions of the nucleus that each component depends on, see the corresponding component topic. For more information about updating the nucleus, see Update the AWS IoT Greengrass Core software (OTA).

Component	Description	Component type	Supported OS	Open source
SageMaker AI Edge Manager	Deploys the Amazon SageMaker AI Edge Manager agent on the Greengrass core device.	Generic	Linux, Windows	No
DLR image classification	Inference component that uses the DLR image classification model store and the DLR runtime component as dependencies to install DLR, download sample image classification models, and perform image classification inference on supported devices.	Generic	Linux, Windows	No
DLR object detection	Inference component that uses the DLR object	Generic	Linux, Windows	No

Component	Description	Component type	Supported OS	Open source
	detection model store and the DLR runtime component as dependencies to install DLR, download sample object detection models, and perform object detection inference on supported devices.			
DLR image classification model store	Model component that contains sample ResNet-50 image classification models as Greengrass artifacts.	Generic	Linux, Windows	No

Component	Description	Component type	Supported OS	Open source
DLR object detection model store	Model component that contains sample YOLOv3 object detection models as Greengrass artifacts.	Generic	Linux, Windows	No
DLR runtime	Runtime component that contains an installation script that is used to install DLR and its dependenc ies on the Greengrass core device.	Generic	Linux, Windows	No

Component	Description	Component type	Supported OS	Open source
TensorFlow Lite image classification	Inference component that uses the TensorFlow Lite image classific ation model store and the TensorFlow Lite runtime component as dependenc ies to install TensorFlow Lite, download sample image classification models, and perform image classification inference on supported devices.	Generic	Linux, Windows	No

Component	Description	Component type	Supported OS	Open source
TensorFlow Lite object detection	Inference component that uses the TensorFlow Lite object detection model store and the TensorFlow Lite runtime component as dependencies to install TensorFlow Lite, download sample object detection models, and perform object detection inference on supported devices.	Generic	Linux, Windows	No
TensorFlow Lite image classification model store	Model component that contains a sample MobileNet v1 model as a Greengrass artifact.	Generic	Linux, Windows	No

Component	Description	Component type	Supported OS	Open source
TensorFlow Lite object detection model store	Model component that contains a sample Single Shot Detection (SSD) MobileNet model as a Greengrass artifact.	Generic	Linux, Windows	No
TensorFlow Lite runtime	Runtime component that contains an installation script that is used to install TensorFlow Lite and its dependencies on the Greengrass core device.	Generic	Linux, Windows	No

Use Amazon SageMaker AI Edge Manager on Greengrass core devices

Important

SageMaker AI Edge Manager was discontinued on April 26th, 2024. For more information about continuing to deploy your models to edge devices, see SageMaker AI Edge Manager end of life.

Amazon SageMaker AI Edge Manager is a software agent that runs on edge devices. SageMaker AI Edge Manager provides model management for edge devices so that you can package and use Amazon SageMaker AI Neo-compiled models directly on Greengrass core devices. By using SageMaker AI Edge Manager, you can also sample model input and output data from your core devices, and send that data to the AWS Cloud for monitoring and analysis. Because SageMaker AI Edge Manager uses SageMaker AI Neo to optimize your models for your target hardware, you don't need to install the DLR runtime directly on your device. On Greengrass devices, SageMaker AI Edge Manager doesn't load local AWS IoT certificates or call the AWS IoT credential provider endpoint directly. Instead, SageMaker AI Edge Manager uses the token exchange service to fetch temporary credential from a TES endpoint.

This section describes how SageMaker AI Edge Manager works on Greengrass core devices.

How SageMaker AI Edge Manager works on Greengrass devices

To deploy the SageMaker AI Edge Manager agent to your core devices, create a deployment that includes the `aws.greengrass.SageMakerEdgeManager` component. AWS IoT Greengrass manages the installation and lifecycle of the Edge Manager agent on your devices. When a new version of the agent binary is available, deploy the updated version of the `aws.greengrass.SageMakerEdgeManager` component to upgrade the version of the agent that is installed on your device.

When you use SageMaker AI Edge Manager with AWS IoT Greengrass, your workflow includes the following high-level steps:

1. Compile models with SageMaker AI Neo.

2. Package your SageMaker AI Neo-compiled models using SageMaker AI edge packaging jobs. When you run an edge packaging job for your model, you can choose to create a model component with the packaged model as an artifact that can be deployed to your Greengrass core device.
3. Create a custom inference component. You use this inference component to interact with the Edge Manager agent to perform inference on the core device. These operations include loading models, invoke prediction requests to run inference, and unloading models when the component shuts down.
4. Deploy the SageMaker AI Edge Manager component, the packaged model component, and the inference component to run your model on the SageMaker AI inference engine (Edge Manager agent) on your device.

For more information about creating edge packaging jobs and inference components that work with SageMaker AI Edge Manager, see Deploy Model Package and Edge Manager Agent with AWS IoT Greengrass in the *Amazon SageMaker AI Developer Guide*.

The Tutorial: Get started with SageMaker AI Edge Manager tutorial shows you how to set up and use the SageMaker AI Edge Manager agent on an existing Greengrass core device, using AWS-provided example code that you can use to create sample inference and model components.

When you use SageMaker AI Edge Manager on Greengrass core devices, you can also use the capture data feature to upload sample data to the AWS Cloud. Capture data is a SageMaker AI feature that you use to upload inference input, inference results, and additional inference data to an S3 bucket or a local directory for future analysis. For more information about using capture data with SageMaker AI Edge Manager, see Manage Model in the *Amazon SageMaker AI Developer Guide*.

Requirements

You must meet the following requirements to use the SageMaker AI Edge Manager agent on Greengrass core devices.

- A Greengrass core device running on Amazon Linux 2, a Debian-based Linux platform (x86_64 or Armv8), or Windows (x86_64). If you don't have one, see Tutorial: Getting started with AWS IoT Greengrass V2.
- Python 3.6 or later, including `pip` for your version of Python, installed on your core device.
- The Greengrass device role configured with the following:

- A trust relationship that allows `credentials.iot.amazonaws.com` and `sagemaker.amazonaws.com` to assume the role, as shown in the following IAM policy example.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "credentials.iot.amazonaws.com"
       },
      "Action": "sts:AssumeRole"
    },
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "sagemaker.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

- The AmazonSageMakerEdgeDeviceFleetPolicy IAM managed policy.
- The `s3:PutObject` action, as shown in the following IAM policy example.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "s3:PutObject"
      ],
      "Resource": [
        "*"
      ],
      "Effect": "Allow"
    }
  ]
}
```

- An Amazon S3 bucket created in the same AWS account and AWS Region as your Greengrass core device. SageMaker AI Edge Manager requires an S3 bucket to create an edge device fleet, and to store sample data from running inference on your device. For information about creating S3 buckets, see Getting started with Amazon S3.
- A SageMaker AI edge device fleet that uses the same AWS IoT role alias as your Greengrass core device. For more information, see Create an edge device fleet.
- Your Greengrass core device registered as an edge device in your SageMaker AI Edge device fleet. The edge device name must match the AWS IoT thing name for your core device. For more information, see Register your Greengrass core device.

Get started with SageMaker AI Edge Manager

You can complete a tutorial to get started using SageMaker AI Edge Manager. The tutorial shows you how to get started using SageMaker AI Edge Manager with AWS-provided sample components on an existing core device. These sample components use the SageMaker AI Edge Manager component as a dependency to deploy the Edge Manager agent, and perform inference using pre-trained models that were compiled using SageMaker AI Neo. For more information, see Tutorial: Get started with SageMaker AI Edge Manager.

Customize your machine learning components

In AWS IoT Greengrass, you can configure sample machine learning components to customize how you perform machine learning inference on your devices with the inference, model, and runtime components as the building blocks. AWS IoT Greengrass also provides you the flexibility to use the sample components as templates and create your own custom components as needed. You can mix and match this modular approach to customize your machine learning inference components in the following ways:

Using sample inference components

- Modify the configuration of inference components when you deploy them.
- Use a custom model with the sample inference component by replacing the sample model store component with a custom model component. Your custom model must be trained using the same runtime as the sample model.

Using custom inference components

- Use custom inference code with the sample models and runtimes by adding public model components and runtime components as dependencies of custom inference components.

- Create and add custom model components or runtime components as dependencies of custom inference components. You must use custom components if you want to use custom inference code or a runtime for which AWS IoT Greengrass doesn't provide a sample component.

Topics

- Modify the configuration of a public inference component
- Use a custom model with the sample inference component
- Create custom machine learning components
- Create a custom inference component

Modify the configuration of a public inference component

In the AWS IoT Greengrass console, the component page displays the default configuration of that component. For example, the default configuration of the TensorFlow Lite image classification component looks like the following:

```
{
  "accessControl": {
    "aws.greengrass.ipc.mqttproxy": {
      "aws.greengrass.TensorFlowLiteImageClassification:mqttproxy:1": {
        "policyDescription": "Allows access to publish via topic ml/tflite/image-
classification.",
        "operations": [
          "aws.greengrass#PublishToIoTCore"
        ],
        "resources": [
          "ml/tflite/image-classification"
        ]
      }
    }
  },
  "PublishResultsOnTopic": "ml/tflite/image-classification",
  "ImageName": "cat.jpeg",
  "InferenceInterval": 3600,
  "ModelResourceKey": {
    "model": "TensorFlowLite-Mobilenet"
  }
```

```
}
```

When you deploy a public inference component, you can modify the default configuration to customize your deployment. For information about the available configuration parameters for each public inference component, see the component topic in AWS-provided machine learning components.

This section describes how to deploy a modified component from the AWS IoT Greengrass console. For information about deploying components using the AWS CLI, see Create deployments.

To deploy a modified public inference component (console)

1. Sign in to the AWS IoT Greengrass console.
2. In the navigation menu, choose **Components**.
3. On the **Components** page, on the **Public components** tab, choose the component you want to deploy.
4. On the component page, choose **Deploy**.
5. From **Add to deployment**, choose one of the following:
   a. To merge this component to an existing deployment on your target device, choose **Add to existing deployment**, and then select the deployment that you want to revise.
   b. To create a new deployment on your target device, choose **Create new deployment**. If you have an existing deployment on your device, choosing this step replaces the existing deployment.
6. On the **Specify target** page, do the following:
   a. Under **Deployment** information, enter or modify the friendly name for your deployment.
   b. Under **Deployment targets**, select a target for your deployment, and choose **Next**. You cannot change the deployment target if you are revising an existing deployment.
7. On the **Select components** page, under **Public components** verify that the inference component with your modified configuration is selected, and choose **Next**.
8. On the **Configure components** page, do the following:
   a. Select the inference component, and choose **Configure component**.
   b. Under **Configuration update**, enter the configuration values that you want to update. For example, enter the following configuration update in the **Configuration to merge** box to

change the inference interval to 15 seconds, and instruct the component to look for the image named `custom.jpg` in the `/custom-ml-inference/images/` folder.

```
{
  "InferenceInterval": "15",
  "ImageName": "custom.jpg",
  "ImageDirectory": "/custom-ml-inference/images/"
}
```

To reset a component's entire configuration to its default values, specify a single empty string `""` in the **Reset paths** box.

c. Choose **Confirm**, and then choose **Next**.

9. On the **Configure advanced setting** page, keep the default configuration settings, and choose **Next**.
10. On the **Review** page, choose **Deploy**

Use a custom model with the sample inference component

If you want to use the sample inference component with your own machine learning models for a runtime for which AWS IoT Greengrass provides a sample runtime component, you must override the public model components with components that use those models as artifacts. At a high-level you complete the following steps to use a custom model with the sample inference component:

1. Create a model component that uses a custom model in an S3 bucket as an artifact. Your custom model must be trained using the same runtime as the model that you want to replace.
2. Modify the `ModelResourceKey` configuration parameter in the inference component to use the custom model. For information about updating the configuration of the inference component, see Modify the configuration of a public inference component

When you deploy the inference component, AWS IoT Greengrass looks for the latest version of its component dependencies. It overrides the dependent public model component if a later custom version of the component exists in the same AWS account and AWS Region.

Create a custom model component (console)

1. Upload your model to an S3 bucket. For information about uploading your models to an S3 bucket, see Working with Amazon S3 Buckets in the *Amazon Simple Storage Service User Guide*.

> **Note**
>
> You must store your artifacts in S3 buckets that are in the same AWS account and AWS Region as the components. To enable AWS IoT Greengrass to access these artifacts, the Greengrass device role must allow the `s3:GetObject` action. For more information about the device role, see Authorize core devices to interact with AWS services.

2. In the AWS IoT Greengrass console navigation menu, choose **Components**.
3. Retrieve the component recipe for the public model store component.
   a. On the **Components** page, on the **Public components** tab, look for and choose the public model component for which you want to create a new version. For example, `variant.DLR.ImageClassification.ModelStore`.
   b. On the component page, choose **View recipe** and copy the displayed JSON recipe.
4. On the **Components** page, on the **My components** tab, choose **Create component**.
5. On the **Create component** page, under **Component information**, select **Enter recipe as JSON** as your component source.
6. In the **Recipe** box, paste the component recipe that you previously copied.
7. In the recipe, update the following values:
   - `ComponentVersion`: Increment the minor version of the component.

     When you create a custom component to override a public model component, you must update only the minor version of the existing component version. For example, if the public component version is `2.1.0`, you can create a custom component with version `2.1.1`.
   - `Manifests.Artifacts.Uri`: Update each URI value to the Amazon S3 URI of the model that you want to use.

> **Note**
>
> Do not change the name of the component.

8. Choose **Create component**.

Create a custom model component (AWS CLI)

1. Upload your model to an S3 bucket. For information about uploading your models to an S3 bucket, see Working with Amazon S3 Buckets in the *Amazon Simple Storage Service User Guide*.

   > **Note**
   >
   > You must store your artifacts in S3 buckets that are in the same AWS account and AWS Region as the components. To enable AWS IoT Greengrass to access these artifacts, the Greengrass device role must allow the `s3:GetObject` action. For more information about the device role, see Authorize core devices to interact with AWS services.

2. Run the following command to retrieve the component recipe of the public component. This command writes the component recipe to the output file that you provide in your command. Convert the retrieved base64-encoded string to JSON or YAML, as needed.

   Linux, macOS, or Unix

   ```
   aws greengrassv2 get-component \
       --arn <arn> \
       --recipe-output-format <recipe-format> \
       --query recipe \
       --output text | base64 --decode > <recipe-file>
   ```

   Windows Command Prompt (CMD)

   ```
   aws greengrassv2 get-component ^
       --arn <arn> ^
       --recipe-output-format <recipe-format> ^
       --query recipe ^
       --output text > <recipe-file>.base64

   certutil -decode <recipe-file>.base64 <recipe-file>
   ```

   PowerShell

   ```
   aws greengrassv2 get-component `
       --arn <arn> `
       --recipe-output-format <recipe-format> `
       --query recipe `
   ```

```
    --output text > <recipe-file>.base64

certutil -decode <recipe-file>.base64 <recipe-file>
```

3. Update the name of the recipe file to *<component-name>*-*<component-version>*, where component version is the target version of the new component. For example, `variant.DLR.ImageClassification.ModelStore-2.1.1.yaml`.
4. In the recipe, update the following values:

   - `ComponentVersion`: Increment the minor version of the component.

     When you create a custom component to override a public model component, you must update only the minor version of the existing component version. For example, if the public component version is `2.1.0`, you can create a custom component with version `2.1.1`.
   - `Manifests.Artifacts.Uri`: Update each URI value to the Amazon S3 URI of the model that you want to use.

   > **Note**
   >
   > Do not change the name of the component.

5. Run the following command to create a new component using the recipe you retrieved and modified.

   ```
   aws greengrassv2 create-component-version \
       --inline-recipe fileb://path/to/component/recipe
   ```

   > **Note**
   >
   > This step creates the component in the AWS IoT Greengrass service in the AWS Cloud. You can use the Greengrass CLI to develop, test, and deploy your component locally before you upload it to the cloud. For more information, see Develop AWS IoT Greengrass components.

For more information about creating components, see Develop AWS IoT Greengrass components.

Create custom machine learning components

You must create custom components if you want to use custom inference code or a runtime for which AWS IoT Greengrass doesn't provide a sample component. You can use your custom inference code with the AWS-provided sample machine learning models and runtimes, or you can develop a completely customized machine learning inference solution with your own models and runtime. If your models use a runtime for which AWS IoT Greengrass provides a sample runtime component, then you can use that runtime component, and you need to create custom components only for your inference code and the models you want to use.

Topics

- Retrieve the recipe for a public component
- Retrieve sample component artifacts
- Upload component artifacts to an S3 bucket
- Create custom components

Retrieve the recipe for a public component

You can use the recipe of an existing public machine learning component as a template to create a custom component. To view the component recipe for the latest version of a public component, use the console or the AWS CLI as follows:

- **Using the console**

  1. On the **Components** page, on the **Public components** tab, look for and choose the public component.
  2. On the component page, choose **View recipe**.

- **Using AWS CLI**

  Run the following command to retrieve the component recipe of the public variant component. This command writes the component recipe to the JSON or YAML recipe file that you provide in your command.

  Linux, macOS, or Unix

```
aws greengrassv2 get-component \
    --arn <arn> \
    --recipe-output-format <recipe-format> \
```

```
    --query recipe \
    --output text | base64 --decode > <recipe-file>
```

Windows Command Prompt (CMD)

```
aws greengrassv2 get-component ^
    --arn <arn> ^
    --recipe-output-format <recipe-format> ^
    --query recipe ^
    --output text > <recipe-file>.base64

certutil -decode <recipe-file>.base64 <recipe-file>
```

PowerShell

```
aws greengrassv2 get-component `
    --arn <arn> `
    --recipe-output-format <recipe-format> `
    --query recipe `
    --output text > <recipe-file>.base64

certutil -decode <recipe-file>.base64 <recipe-file>
```

Replace the values in your command as follows:

- *<arn>*. The Amazon Resource Name (ARN) of the public component.
- *<recipe-format>*. The format in which you want to create the recipe file. Supported values are `JSON` and `YAML`.
- *<recipe-file>*. The name of the recipe in the format *<component-name>*-*<component-version>*.

Retrieve sample component artifacts

You can use the artifacts used by the public machine learning components as templates to create your custom component artifacts, such as inference code or runtime installation scripts.

To view the sample artifacts that are included in the public machine learning components, deploy the public inference component and then view the artifacts on your device in the */greengrass/v2*`/packages/artifacts-unarchived/`*component-name*/*component-version*/ folder.

Upload component artifacts to an S3 bucket

Before you can create a custom component, you must upload the component artifacts to an S3 bucket and use the S3 URIs in your component recipe. For example, to use custom inference code in your inference component, upload the code to an S3 bucket. You can then use the Amazon S3 URI of your inference code as an artifact in your component.

For information about uploading content to an S3 bucket, see Working with Amazon S3 Buckets in the *Amazon Simple Storage Service User Guide*.

> **Note**
>
> You must store your artifacts in S3 buckets that are in the same AWS account and AWS Region as the components. To enable AWS IoT Greengrass to access these artifacts, the Greengrass device role must allow the `s3:GetObject` action. For more information about the device role, see Authorize core devices to interact with AWS services.

Create custom components

You can use the artifacts and recipes that you retrieved to create your custom machine learning components. For an example, see Create a custom inference component.

For detailed information about creating and deploying components to Greengrass devices, see Develop AWS IoT Greengrass components and Deploy AWS IoT Greengrass components to devices.

Create a custom inference component

This section shows you how to create a custom inference component using the DLR image classification component as a template.

Topics

- Upload your inference code to an Amazon S3 bucket
- Create a recipe for your inference component
- Create the inference component

Upload your inference code to an Amazon S3 bucket

Create your inference code and then upload it to an S3 bucket. For information about uploading content to an S3 bucket, see Working with Amazon S3 Buckets in the *Amazon Simple Storage Service User Guide*.

> **Note**
>
> You must store your artifacts in S3 buckets that are in the same AWS account and AWS Region as the components. To enable AWS IoT Greengrass to access these artifacts, the Greengrass device role must allow the `s3:GetObject` action. For more information about the device role, see Authorize core devices to interact with AWS services.

Create a recipe for your inference component

1. Run the following command to retrieve the component recipe of the DLR image classification component. This command writes the component recipe to the JSON or YAML recipe file that you provide in your command.

   Linux, macOS, or Unix

   ```
   aws greengrassv2 get-component \
       --arn
    arn:aws:greengrass:region:aws:components:aws.greengrass.DLRImageClassification:versions
    \
       --recipe-output-format JSON | YAML \
       --query recipe \
       --output text | base64 --decode > <recipe-file>
   ```

   Windows Command Prompt (CMD)

   ```
   aws greengrassv2 get-component ^
       --arn
    arn:aws:greengrass:region:aws:components:aws.greengrass.DLRImageClassification:versions
    ^
       --recipe-output-format JSON | YAML ^
       --query recipe ^
       --output text > <recipe-file>.base64
   ```

```
certutil -decode <recipe-file>.base64 <recipe-file>
```

PowerShell

```
aws greengrassv2 get-component `
    --arn
 arn:aws:greengrass:region:aws:components:aws.greengrass.DLRImageClassification:versions
 `
    --recipe-output-format JSON | YAML `
    --query recipe `
    --output text > <recipe-file>.base64

certutil -decode <recipe-file>.base64 <recipe-file>
```

Replace *<recipe-file>* with the name of the recipe in the format *<component-name>*-*<component-version>*.

2. In the `ComponentDependencies` object in your recipe, do one or more of the following depending on the model and runtime components that you want to use:

   - Keep the DLR component dependency if you want to use DLR-compiled models. You can also replace it with a dependency on a custom runtime component, as shown in the following example.

     Runtime component

     JSON

     ```
     {
         "<runtime-component>": {
             "VersionRequirement": "<version>",
             "DependencyType": "HARD"
         }
     }
     ```

     YAML

     ```
     <runtime-component>:
         VersionRequirement: "<version>"
         DependencyType: HARD
     ```

- Keep the DLR image classification model store dependency to use the pre-trained ResNet-50 models that AWS provides, or modify it to use a custom model component. When you include a dependency for a public model component, if a later custom version of the component exists in the same AWS account and AWS Region, then the inference component uses that custom component. Specify the model component dependency as shown in the following examples.

  Public model component

  JSON

  ```
  {
      "variant.DLR.ImageClassification.ModelStore": {
          "VersionRequirement": "<version>",
          "DependencyType": "HARD"
      }
  }
  ```

  YAML

  ```
  variant.DLR.ImageClassification.ModelStore:
      VersionRequirement: "<version>"
      DependencyType: HARD
  ```

  Custom model component

  JSON

  ```
  {
      "<custom-model-component>": {
          "VersionRequirement": "<version>",
          "DependencyType": "HARD"
      }
  }
  ```

  YAML

  ```
  <custom-model-component>:
      VersionRequirement: "<version>"
      DependencyType: HARD
  ```

3. In the `ComponentConfiguration` object, add the default configuration for this component. You can later modify this configuration when you deploy the component. The following excerpt shows the component configuration for the DLR image classification component.

   For example, if you use a custom model component as a dependency for your custom inference component, then modify `ModelResourceKey` to provide the names of the models that you are using.

   JSON

   ```
   {
     "accessControl": {
       "aws.greengrass.ipc.mqttproxy": {
         "aws.greengrass.ImageClassification:mqttproxy:1": {
           "policyDescription": "Allows access to publish via topic ml/dlr/image-
   classification.",
           "operations": [
             "aws.greengrass#PublishToIoTCore"
           ],
           "resources": [
             "ml/dlr/image-classification"
           ]
         }
       }
     },
     "PublishResultsOnTopic": "ml/dlr/image-classification",
     "ImageName": "cat.jpeg",
     "InferenceInterval": 3600,
     "ModelResourceKey": {
       "armv7l": "DLR-resnet50-armv7l-cpu-ImageClassification",
       "x86_64": "DLR-resnet50-x86_64-cpu-ImageClassification",
       "aarch64": "DLR-resnet50-aarch64-cpu-ImageClassification"
     }
   }
   ```

   YAML

   ```
   accessControl:
       aws.greengrass.ipc.mqttproxy:
           'aws.greengrass.ImageClassification:mqttproxy:1':
               policyDescription: 'Allows access to publish via topic ml/dlr/image-
   classification.'
   ```

```
            operations:
                - 'aws.greengrass#PublishToIoTCore'
            resources:
                - ml/dlr/image-classification
PublishResultsOnTopic: ml/dlr/image-classification
ImageName: cat.jpeg
InferenceInterval: 3600
ModelResourceKey:
    armv7l: "DLR-resnet50-armv7l-cpu-ImageClassification"
    x86_64: "DLR-resnet50-x86_64-cpu-ImageClassification"
    aarch64: "DLR-resnet50-aarch64-cpu-ImageClassification"
```

4. In the `Manifests` object, provide information about the artifacts and the configuration of this component that are used when the component is deployed to different platforms and any other information required to successfully run the component. The following excerpt shows the configuration of the `Manifests` object for Linux platform in the DLR image classification component.

   JSON

   ```
   {
     "Manifests": [
       {
         "Platform": {
           "os": "linux",
           "architecture": "arm"
         },
         "Name": "32-bit armv7l - Linux (raspberry pi)",
         "Artifacts": [
           {
             "URI": "s3://SAMPLE-BUCKET/sample-artifacts-directory/
   image_classification.zip",
             "Unarchive": "ZIP"
           }
         ],
         "Lifecycle": {
           "Setenv": {
             "DLR_IC_MODEL_DIR":
    "{variant.DLR.ImageClassification.ModelStore:artifacts:decompressedPath}/
   {configuration:/ModelResourceKey/armv7l}",
             "DEFAULT_DLR_IC_IMAGE_DIR": "{artifacts:decompressedPath}/
   image_classification/sample_images/"
           },
   ```

```
        "Run": {
          "RequiresPrivilege": true,
          "script": ". {variant.DLR:configuration:/MLRootPath}/
greengrass_ml_dlr_venv/bin/activate\npython3 {artifacts:decompressedPath}/
image_classification/inference.py"
        }
      }
    }
  ]
}
```

YAML

```
Manifests:
  - Platform:
      os: linux
      architecture: arm
    Name: 32-bit armv7l - Linux (raspberry pi)
    Artifacts:
      - URI: s3://SAMPLE-BUCKET/sample-artifacts-directory/
image_classification.zip
        Unarchive: ZIP
    Lifecycle:
      SetEnv:
        DLR_IC_MODEL_DIR:
 "{variant.DLR.ImageClassification.ModelStore:artifacts:decompressedPath}/
{configuration:/ModelResourceKey/armv7l}"
        DEFAULT_DLR_IC_IMAGE_DIR: "{artifacts:decompressedPath}/
image_classification/sample_images/"
      Run:
        RequiresPrivilege: true
        script: |-
          . {variant.DLR:configuration:/MLRootPath}/greengrass_ml_dlr_venv/bin/
activate
          python3 {artifacts:decompressedPath}/image_classification/inference.py
```

For detailed information about creating component recipes, see AWS IoT Greengrass component recipe reference.

Create the inference component

Use the AWS IoT Greengrass console or the AWS CLI to create a component using the recipe you just defined. After you create the component, you can deploy it to perform inference on your device. For an example of how to deploy an inference component, see Tutorial: Perform sample image classification inference using TensorFlow Lite.

Create custom inference component (console)

1. Sign in to the AWS IoT Greengrass console.
2. In the navigation menu, choose **Components**.
3. On the **Components** page, on the **My components** tab, choose **Create component**.
4. On the **Create component** page, under **Component information**, select either **Enter recipe as JSON** or **Enter recipe as YAML** as your component source.
5. In the **Recipe** box, enter the custom recipe that you created.
6. Click **Create component**.

Create custom inference component (AWS CLI)

Run the following command to create a new custom component using the recipe that you created.

```
aws greengrassv2 create-component-version \
    --inline-recipe fileb://path/to/recipe/file
```

> **Note**
>
> This step creates the component in the AWS IoT Greengrass service in the AWS Cloud. You can use the Greengrass CLI to develop, test, and deploy your component locally before you upload it to the cloud. For more information, see Develop AWS IoT Greengrass components.

Troubleshooting machine learning inference

Use the troubleshooting information and solutions in this section to help resolve issues with your machine learning components. For the public machine learning inference components, see the error messages in the following component logs:

Linux or Unix

- */greengrass/v2*/logs/aws.greengrass.DLRImageClassification.log
- */greengrass/v2*/logs/aws.greengrass.DLRObjectDetection.log
- */greengrass/v2*/logs/aws.greengrass.TensorFlowLiteImageClassification.log
- */greengrass/v2*/logs/aws.greengrass.TensorFlowLiteObjectDetection.log

Windows

- *C:\greengrass\v2*\logs\aws.greengrass.DLRImageClassification.log
- *C:\greengrass\v2*\logs\aws.greengrass.DLRObjectDetection.log
- *C:\greengrass\v2*\logs\aws.greengrass.TensorFlowLiteImageClassification.log
- *C:\greengrass\v2*\logs\aws.greengrass.TensorFlowLiteObjectDetection.log

If a component is installed correctly, then the component log contains the location of the library that it uses for inference.

Issues

- Failed to fetch library
- Cannot open shared object file
- Error: ModuleNotFoundError: No module named '<library>'
- No CUDA-capable device is detected
- No such file or directory
- RuntimeError: module compiled against API version 0xf but this version of NumPy is <version>
- picamera.exc.PiCameraError: Camera is not enabled
- Memory errors
- Disk space errors
- Timeout errors

Failed to fetch library

The following error occurs when the installer script fails to download a required library during deployment on a Raspberry Pi device.

```
Err:2 http://raspbian.raspberrypi.org/raspbian buster/main armhf python3.7-dev armhf
 3.7.3-2+deb10u1
404 Not Found [IP: 93.93.128.193 80]
E: Failed to fetch http://raspbian.raspberrypi.org/raspbian/pool/main/p/python3.7/
libpython3.7-dev_3.7.3-2+deb10u1_armhf.deb 404 Not Found [IP: 93.93.128.193 80]
```

Run `sudo apt-get update` and deploy your component again.

Cannot open shared object file

You might see errors similar to the following when the installer script fails to download a required dependency for `opencv-python` during deployment on a Raspberry Pi device.

```
ImportError: libopenjp2.so.7: cannot open shared object file: No such file or directory
```

Run the following command to manually install the dependencies for `opencv-python`:

```
sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev
 libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
```

Error: ModuleNotFoundError: No module named '<library>'

You might see this error in the ML runtime component logs (`variant.DLR.log` or `variant.TensorFlowLite.log`) when the ML runtime library or its dependencies aren't installed correctly. This error can occur in the following cases:

- If you use the `UseInstaller` option, which is enabled by default, this error indicates that the ML runtime component failed to install the runtime or its dependencies. Do the following:
  1. Configure the ML runtime component to disable the `UseInstaller` option.
  2. Install the ML runtime and its dependencies, and make them available to the system user that runs the ML components. For more information, see the following:
     - DLR runtime UseInstaller option

  - TensorFlow Lite runtime UseInstaller option

- If you don't use the `UseInstaller` option, this error indicates that the ML runtime or its dependencies aren't installed for the system user that runs the ML components. Do the following:

  1. Check that the library is installed for the system user that runs the ML components. Replace *ggc_user* with the name of the system user, and replace *tflite_runtime* with the name of the library to check.

     Linux or Unix

     ```
     sudo -H -u ggc_user bash -c "python3 -c 'import tflite_runtime'"
     ```

     Windows

     ```
     runas /user:ggc_user "py -3 -c \"import tflite_runtime\""
     ```

  2. If the library isn't installed, install it for that user. Replace *ggc_user* with the name of the system user, and replace *tflite_runtime* with the name of the library.

     Linux or Unix

     ```
     sudo -H -u ggc_user bash -c "python3 -m pip install --user tflite_runtime"
     ```

     Windows

     ```
     runas /user:ggc_user "py -3 -m pip install --user tflite_runtime"
     ```

     For more information about the dependencies for each ML runtime, see the following:

     - DLR runtime UseInstaller option
     - TensorFlow Lite runtime UseInstaller option

  3. If the issue persists, install the library for another user to confirm whether this device can install the library. The user could be, for example, your user, the root user, or an administrator user. If you can't install the library successfully for any user, your device might not support the library. Consult the library's documentation to review requirements and troubleshoot installation issues.

No CUDA-capable device is detected

You might see the following error when you use GPU acceleration. Run the following command to enable GPU access for the Greengrass user.

```
sudo usermod -a -G video ggc_user
```

No such file or directory

The following errors indicate that the runtime component was unable to set up the virtual environment correctly:

- *MLRootPath*/greengrass_ml_dlr_conda/bin/conda: No such file or directory
- *MLRootPath*/greengrass_ml_dlr_venv/bin/activate: No such file or directory
- *MLRootPath*/greengrass_ml_tflite_conda/bin/conda: No such file or directory
- *MLRootPath*/greengrass_ml_tflite_venv/bin/activate: No such file or directory

Check the logs to make sure that all runtime dependencies were installed correctly. For more information about the libraries installed by the installer script, see the following topics:

- DLR runtime
- TensorFlow Lite runtime

By default *MLRootPath* is set to */greengrass/v2*/work/*component-name*/greengrass_ml. To change this location, include the DLR runtime or TensorFlow Lite runtime runtime component directly in your deployment, and specify a modified value for the MLRootPath parameter in a configuration merge update. For more information about configuring component, see Update component configurations.

> **Note**
>
> For the DLR component v1.3.x, you set the MLRootPath parameter in the configuration of the inference component, and the default value is $HOME/greengrass_ml.

RuntimeError: module compiled against API version 0xf but this version of NumPy is <version>

You might see the following errors when you run machine learning inference on a Raspberry Pi running Raspberry Pi OS Bullseye.

```
RuntimeError: module compiled against API version 0xf but this version of numpy is 0xd
ImportError: numpy.core.multiarray failed to import
```

This error occurs because Raspberry Pi OS Bullseye includes an earlier version of NumPy than the version that OpenCV requires. To fix this issue, run the following command to upgrade NumPy to the latest version.

```
pip3 install --upgrade numpy
```

picamera.exc.PiCameraError: Camera is not enabled

You might see the following error when you run machine learning inference on a Raspberry Pi running Raspberry Pi OS Bullseye.

```
picamera.exc.PiCameraError: Camera is not enabled. Try running 'sudo raspi-config' and
 ensure that the camera has been enabled.
```

This error occurs because Raspberry Pi OS Bullseye includes a new camera stack that isn't compatible with the ML components. To fix this issue, enable the legacy camera stack.

To enable the legacy camera stack

1. Run the following command to open the Raspberry Pi configuration tool.

   ```
   sudo raspi-config
   ```

2. Select **Interface Options**.
3. Select **Legacy camera** to enable the legacy camera stack.
4. Reboot the Raspberry Pi.

Memory errors

The following errors typically occur when the device does not have enough memory and the component process is interrupted.

- `stderr. Killed.`
- `exitCode=137`

We recommend a minimum of 500 MB of memory to deploy a public machine learning inference component.

Disk space errors

The `no space left on device` error typically occurs when a device does not have enough storage. Make sure that there is enough disk space available on your device before you deploy the component again. We recommend a minimum of 500 MB of free disk space to deploy a public machine learning inference component.

Timeout errors

The public machine learning components download large machine learning model files that are larger than 200 MB. If the download times out during deployment, check your internet connection speed and retry the deployment.

Manage Greengrass core devices with AWS Systems Manager

> **Note**
>
> AWS IoT Greengrass doesn't currently support this feature on Windows core devices.

Systems Manager is an AWS service that you can use to view and control your infrastructure on AWS, including Amazon EC2 instances, on-premises servers and virtual machines (VMs), and edge devices. Systems Manager enables you to view operational data, automate operation tasks, and maintain security and compliance. When you register a machine with Systems Manager, it's called a *managed node*. For more information, see What is AWS Systems Manager? in the *AWS Systems Manager User Guide*.

The AWS Systems Manager Agent (Systems Manager Agent) is software that you can install on devices to enable Systems Manager to update, manage, and configure them. To install the Systems Manager Agent on Greengrass core devices, deploy the Systems Manager Agent component. When you deploy the Systems Manager Agent for the first time, it registers the core device as a Systems Manager managed node. The Systems Manager Agent runs on the device to enable communication with the Systems Manager service in the AWS Cloud. For more information about how to install and configure the Systems Manager Agent component, see Install the AWS Systems Manager Agent.

Systems Manager tools and features are called *capabilities*. Greengrass core devices support all Systems Manager capabilities. For more information about these capabilities and how to use Systems Manager to manage core devices, see Systems Manager capabilities in the *AWS Systems Manager User Guide*.

AWS Systems Manager offers a standard-instances tier and an advanced-instances tier for Systems Manager managed nodes. If you're using Systems Manager for the first time, you start on the standard-instances tier. On the standard-instances tier, you can register up to 1,000 managed nodes per AWS Region in your AWS account. If you need to register more than 1,000 managed nodes in a single account and Region, or if you need to use the Session Manager capability, use the advanced-instances tier. For more information, see Configuring instance tiers in the *AWS Systems Manager User Guide*.

Topics

- Install the AWS Systems Manager Agent
- Uninstall the AWS Systems Manager Agent

Install the AWS Systems Manager Agent

The AWS Systems Manager Agent (Systems Manager Agent) is Amazon software that you install to enable Systems Manager to update, manage, and configure Greengrass core devices, Amazon EC2 instances, and other resources. The agent processes and runs requests from the Systems Manager service in the AWS Cloud. Then, the agent sends status and runtime information back to the Systems Manager service. For more information, see About Systems Manager Agent in the *AWS Systems Manager User Guide*.

AWS provides the Systems Manager Agent as a Greengrass component that you can deploy to your Greengrass core devices to manage them with Systems Manager. The Systems Manager Agent component installs the Systems Manager Agent software and registers the core device as a managed node in Systems Manager. Follow the steps on this page to complete prerequisites and deploy the Systems Manager Agent component to a core device or group of core devices.

Topics

- Step 1: Complete general Systems Manager setup steps
- Step 2: Create an IAM service role for Systems Manager
- Step 3: Add permissions to the token exchange role
- Step 4: Deploy the Systems Manager Agent component
- Step 5: Verify core device registration with Systems Manager

Step 1: Complete general Systems Manager setup steps

If you haven't already done so, complete general setup steps for AWS Systems Manager. For more information, see Complete general Systems Manager setup steps in the *AWS Systems Manager User Guide*.

Step 2: Create an IAM service role for Systems Manager

The Systems Manager Agent uses an AWS Identity and Access Management (IAM) service role to communicate with AWS Systems Manager. Systems Manager assumes this role to enable Systems

Manager capabilities on each core device. The Systems Manager Agent component also uses this role to register the core device as a Systems Manager managed node when you deploy the component. If you haven't already done so, create a Systems Manager service role for the Systems Manager Agent component to use. For more information, see Create an IAM service role for edge devices in the *AWS Systems Manager User Guide*.

Step 3: Add permissions to the token exchange role

Greengrass core devices use an IAM service role, called the token exchange role, to interact with AWS services. Each core device has a token exchange role that you create when you install the AWS IoT Greengrass Core software. Many Greengrass components, such as the Systems Manager Agent, require additional permissions on this role. The Systems Manager agent component requires the following permissions, which include permission to use the role that you created in Step 2: Create an IAM service role for Systems Manager.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "iam:PassRole"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:iam::account-id:role/SSMServiceRole"
      ]
    },
    {
      "Action": [
        "ssm:AddTagsToResource",
        "ssm:RegisterManagedInstance"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

If you haven't already done so, add these permissions to the core device's token exchange role to allow the Systems Manager Agent to operate. You can add a new policy to the token exchange role to grant this permission.

To add permissions to the token exchange role (console)

1. In the IAM console navigation menu, choose **Roles**.
2. Choose the IAM role that you set up as a token exchange role when you installed the AWS IoT Greengrass Core software. If you didn't specify a name for the token exchange role when you installed the AWS IoT Greengrass Core software, it created a role named `GreengrassV2TokenExchangeRole`.
3. Under **Permissions**, choose **Add permissions**, and then choose **Attach policies**.
4. Choose **Create policy**. The **Create policy** page opens in a new browser tab.
5. On the **Create policy** page, do the following:

   a. Choose **JSON** to open the JSON editor.

   b. Paste the following policy into the JSON editor. Replace *SSMServiceRole* with the name of the service role that you created in Step 2: Create an IAM service role for Systems Manager.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "iam:PassRole"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:iam::account-id:role/SSMServiceRole"
      ]
    },
    {
      "Action": [
        "ssm:AddTagsToResource",
        "ssm:RegisterManagedInstance"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

   c. Choose **Next: Tags**.

d. Choose **Next: Review**.

e. Enter a **Name** for the policy, such as **`GreengrassSSMAgentComponentPolicy`**.

f. Choose **Create policy**.

g. Switch to the previous browser tab where you have the token exchange role open.

6. On the **Add permissions** page, choose the refresh button, and then select the Greengrass Systems Manager agent policy that you created in the previous step.

7. Choose **Attach policies**.

   The core devices that use this token exchange role now have permission to interact with the Systems Manager service.

To add permissions to the token exchange role (AWS CLI)

To add a policy that grants permission to use Systems Manager

1. Create a file called `ssm-agent-component-policy.json` and copy the following JSON into the file. Replace *`SSMServiceRole`* with the name of the service role that you created in Step 2: Create an IAM service role for Systems Manager.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "iam:PassRole"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:iam::account-id:role/SSMServiceRole"
      ]
    },
    {
      "Action": [
        "ssm:AddTagsToResource",
        "ssm:RegisterManagedInstance"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
```

```
  ]
}
```

2. Run the following command to create the policy from the policy document in `ssm-agent-component-policy.json`.

   Linux or Unix

   ```
   aws iam create-policy \
     --policy-name GreengrassSSMAgentComponentPolicy \
     --policy-document file://ssm-agent-component-policy.json
   ```

   Windows Command Prompt (CMD)

   ```
   aws iam create-policy ^
     --policy-name GreengrassSSMAgentComponentPolicy ^
     --policy-document file://ssm-agent-component-policy.json
   ```

   PowerShell

   ```
   aws iam create-policy `
     --policy-name GreengrassSSMAgentComponentPolicy `
     --policy-document file://ssm-agent-component-policy.json
   ```

   Copy the policy Amazon Resource Name (ARN) from the policy metadata in the output. You use this ARN to attach this policy to the core device role in the next step.

3. Run the following command to attach the policy to the token exchange role.

   - Replace *GreengrassV2TokenExchangeRole* with the name of the token exchange role that you specified when you installed the AWS IoT Greengrass Core software. If you didn't specify a name for the token exchange role when you installed the AWS IoT Greengrass Core software, it created a role named `GreengrassV2TokenExchangeRole`.
   - Replace the policy ARN with the ARN from the previous step.

   Linux or Unix

   ```
   aws iam attach-role-policy \
     --role-name GreengrassV2TokenExchangeRole \
   ```

```
  --policy-arn
 arn:aws:iam::123456789012:policy/GreengrassSSMAgentComponentPolicy
```

Windows Command Prompt (CMD)

```
aws iam attach-role-policy ^
  --role-name GreengrassV2TokenExchangeRole ^
  --policy-arn
 arn:aws:iam::123456789012:policy/GreengrassSSMAgentComponentPolicy
```

PowerShell

```
aws iam attach-role-policy `
  --role-name GreengrassV2TokenExchangeRole `
  --policy-arn
 arn:aws:iam::123456789012:policy/GreengrassSSMAgentComponentPolicy
```

If the command has no output, it succeeded. The core devices that use this token exchange role now have permission to interact with the Systems Manager service.

Step 4: Deploy the Systems Manager Agent component

Complete the following steps to deploy and configure the Systems Manager Agent component. You can deploy the component to a single core device or to a group of core devices.

To deploy the Systems Manager Agent component (console)

1. In the AWS IoT Greengrass console navigation menu, choose **Components**.
2. On the **Components** page, choose the **Public components** tab, and then choose **aws.greengrass.SystemsManagerAgent**.
3. On the **aws.greengrass.SystemsManagerAgent** page, choose **Deploy**.
4. From **Add to deployment**, choose an existing deployment to revise, or choose to create a new deployment, and then choose **Next**.
5. If you chose to create a new deployment, choose the target core device or thing group for the deployment. On the **Specify target** page, under **Deployment target**, choose a core device or thing group, and then choose **Next**.

6. On the **Select components** page, verify that the **aws.greengrass.SystemsManagerAgent** component is selected, choose **Next**.
7. On the **Configure components** page, select **aws.greengrass.SystemsManagerAgent**, and then do the following:

   a. Choose **Configure component**.

   b. In the **Configure aws.greengrass.SystemsManagerAgent** modal, under **Configuration update**, in **Configuration to merge**, enter the following configuration update. Replace *SSMServiceRole* with the name of the service role that you created in Step 2: Create an IAM service role for Systems Manager.

   ```
   {
     "SSMRegistrationRole": "SSMServiceRole",
     "SSMOverrideExistingRegistration": false
   }
   ```

   > **Note**
   >
   > If the core device already runs the Systems Manager Agent registered with a hybrid activation, change `SSMOverrideExistingRegistration` to `true`. This parameter specifies whether the Systems Manager Agent component registers the core device when the Systems Manager Agent is already running on the device with a hybrid activation.
   > You can also specify tags (`SSMResourceTags`) to add to the Systems Manager managed node that the Systems Manager Agent component creates for the core device. For more information, see Systems Manager Agent component configuration.

   c. Choose **Confirm** to close the modal, and then choose **Next**.
8. On the **Configure advanced settings** page, keep the default configuration settings, and choose **Next**.
9. On the **Review** page, choose **Deploy**.

   The deployment can take up to a minute to complete.

To deploy the Systems Manager Agent component (AWS CLI)

To deploy the Systems Manager Agent component, create a deployment document that includes `aws.greengrass.SystemsManagerAgent` in the `components` object, and specify the configuration update for the component. Follow instructions in Create deployments to create a new deployment or revise an existing deployment.

The following example partial deployment document specifies to use a service role named `SSMServiceRole`. Replace *`SSMServiceRole`* with the name of the service role that you created in Step 2: Create an IAM service role for Systems Manager.

```
{
  ...,
  "components": {
    ...,
    "aws.greengrass.SystemsManagerAgent": {
      "componentVersion": "1.0.0",
      "configurationUpdate": {
        "merge": "{\"SSMRegistrationRole\":\"SSMServiceRole\",
\"SSMOverrideExistingRegistration\":false}"
      }
    }
  }
}
```

Note

If the core device already runs the Systems Manager Agent registered with a hybrid activation, change `SSMOverrideExistingRegistration` to `true`. This parameter specifies whether the Systems Manager Agent component registers the core device when the Systems Manager Agent is already running on the device with a hybrid activation. You can also specify tags (`SSMResourceTags`) to add to the Systems Manager managed node that the Systems Manager Agent component creates for the core device. For more information, see Systems Manager Agent component configuration.

The deployment can take several minutes to complete. You can use the AWS IoT Greengrass service to check the status of the deployment, and you can check the AWS IoT Greengrass Core software logs and Systems Manager Agent component logs to verify that the Systems Manager Agent runs successfully. For more information, see the following:

- Check deployment status
- Monitor AWS IoT Greengrass logs
- Viewing Systems Manager Agent logs in the *AWS Systems Manager User Guide*

If the deployment fails or the Systems Manager Agent doesn't run, you can troubleshoot the deployment on each core device. For more information, see the following:

- Troubleshooting AWS IoT Greengrass V2
- Troubleshooting Systems Manager Agent in the *AWS Systems Manager User Guide*

Step 5: Verify core device registration with Systems Manager

When the Systems Manager Agent component runs, it registers the core device as a managed node in Systems Manager. You can use the AWS IoT Greengrass console, Systems Manager console, and Systems Manager API to verify that a core device is registered as a managed node. Managed nodes are also called instances in parts of the AWS console and API.

To verify core device registration (AWS IoT Greengrass console)

1. In the AWS IoT Greengrass console navigation menu, choose **Core devices**.
2. Choose the core device to verify.
3. On the core device's details page, find the **AWS Systems Manager instance** property. If this property is present and displays a link to the Systems Manager console, the core device is registered as a managed node.

   You can also find the **AWS Systems Manager ping status** property to check the status of the Systems Manager Agent on the core device. When the status is **Online**, you can manage the core device with Systems Manager.

To verify core device registration (Systems Manager console)

1. In the Systems Manager console navigation menu, choose **Fleet Manager**.
2. Under **Managed nodes**, do the following:

   a. Add a filter where **Source type** is **AWS::IoT::Thing**.

   b. Add a filter where **Source ID** is the name of the core device to verify.

3. Find the core device in the **Managed nodes** table. If the core device is in the table, it's registered as a managed node.

   You can also find the **Systems Manager Agent ping status** property to check the status of the Systems Manager Agent on the core device. When the status is **Online**, you can manage the core device with Systems Manager.

To verify core device registration (AWS CLI)

- Use the DescribeInstanceInformation operation to get the list of managed nodes that match a filter that you specify. Run the following command to verify whether a core device is registered as a managed node. Replace *MyGreengrassCore* with the name of the core device to verify.

  ```
  aws ssm describe-instance-information --filter
   Key=SourceIds,Values=MyGreengrassCore Key=SourceTypes,Values=AWS::IoT::Thing
  ```

  The response contains the list of managed nodes that match the filter. If the list contains a managed node, the core device is registered as a managed node. You can also find other information about the core device's managed node in the response. If the `PingStatus` property is `Online`, you can manage the core device with Systems Manager.

After you verify that a core device is registered as a managed node in Systems Manager, you can use the Systems Manager console and API to manage that core device. For more information about the Systems Manager capabilities that you can use to manage Greengrass core devices, see Systems Manager capabilities in the *AWS Systems Manager User Guide*.

Uninstall the AWS Systems Manager Agent

If you no longer want to manage a Greengrass core device with AWS Systems Manager, you can deregister the core device from Systems Manager and uninstall the AWS Systems Manager Agent (Systems Manager Agent) from the device.

You can reregister a core device again at any time. To do so, deploy the Systems Manager Agent component again, which registers the core device with Systems Manager when it installs. Systems Manager stores the command history for a deregistered core device for 30 days.

Topics

- Step 1: Deregister the core device from Systems Manager
- Step 2: Uninstall the Systems Manager Agent component
- Step 3: Uninstall the Systems Manager Agent software

Step 1: Deregister the core device from Systems Manager

You can use the Systems Manager console or API to deregister the core device. For more information, see Deregistering managed nodes in the *AWS Systems Manager User Guide*.

Step 2: Uninstall the Systems Manager Agent component

After you deregister the core device, uninstall the Systems Manager Agent component from the device. To remove a component from a Greengrass core device, revise the deployment that installed the component, and remove the component from the deployment. The AWS IoT Greengrass Core software uninstalls a component when none of a core device's deployments specify that component. For more information, see Deploy AWS IoT Greengrass components to devices.

To uninstall the Systems Manager Agent component (console)

1. In the AWS IoT Greengrass console navigation menu, choose **Core devices**.
2. Choose the core device where you want to uninstall the Systems Manager Agent component.
3. On the core device details page, choose the **Deployments** tab.
4. Choose the deployment that deploys the Systems Manager Agent component to the core device.
5. On the deployment details page, choose **Revise**.
6. In the **Revise deployment** modal, choose **Revise deployment**.
7. In **Step 1: Specify target**, choose **Next**.
8. In **Step 2: Select components**, clear the selection for the **aws.greengrass.SystemsManagerAgent** component, and then choose **Next**.
9. In **Step 3: Configure components**, choose **Next**.
10. In **Step 4: Configure advanced settings**, choose **Next**.
11. In **Step 5: Review**, choose **Deploy**.

To uninstall the Systems Manager Agent component (CLI)

To uninstall the Systems Manager Agent component, revise the deployment that deploys it, and remove it from the deployment. For more information, see Revise deployments.

The deployment can take several minutes to complete. You can use the AWS IoT Greengrass service to check the status of the deployment. For more information, see Check deployment status.

Step 3: Uninstall the Systems Manager Agent software

The Systems Manager Agent software continues to run on the core device after you remove the Systems Manager Agent component. To remove the Systems Manager Agent software, you can run commands on the core device. For more information, see Uninstall Systems Manager Agent from Linux instances in the *AWS Systems Manager User Guide*.

Security in AWS IoT Greengrass

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS Compliance Programs. To learn about the compliance programs that apply to AWS IoT Greengrass, see AWS Services in Scope by Compliance Program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors, including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

When you use AWS IoT Greengrass, you are also responsible for securing your devices, local network connection, and private keys.

This documentation helps you understand how to apply the shared responsibility model when using AWS IoT Greengrass. The following topics show you how to configure AWS IoT Greengrass to meet your security and compliance objectives. You also learn how to use other AWS services that help you to monitor and secure your AWS IoT Greengrass resources.

Topics

- Data protection in AWS IoT Greengrass
- Device authentication and authorization for AWS IoT Greengrass
- Identity and access management for AWS IoT Greengrass
- Allow device traffic through a proxy or firewall
- Compliance validation for AWS IoT Greengrass
- FIPS endpoints
- Resilience in AWS IoT Greengrass
- Infrastructure security in AWS IoT Greengrass

- Configuration and vulnerability analysis in AWS IoT Greengrass
- Code integrity in AWS IoT Greengrass V2
- AWS IoT Greengrass and interface VPC endpoints (AWS PrivateLink)
- Security best practices for AWS IoT Greengrass
- Run AWS IoT Greengrass V2 as a non-root user

Data protection in AWS IoT Greengrass

The AWS shared responsibility model applies to data protection in AWS IoT Greengrass. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. You are also responsible for the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see Data Privacy FAQ. For information about data protection in Europe, see the General Data Protection Regulation (GDPR) Center.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual users with AWS IAM Identity Center or AWS Identity and Access Management (IAM). That way, each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We require TLS 1.2 and recommend TLS 1.3.
- Set up API and user activity logging with AWS CloudTrail. For information about using CloudTrail trails to capture AWS activities, see Working with CloudTrail trails in the *AWS CloudTrail User Guide*.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing sensitive data that is stored in Amazon S3.
- If you require FIPS 140-3 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see Federal Information Processing Standard (FIPS) 140-3.

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form text fields such as a **Name** field. This includes

when you work with AWS IoT Greengrass or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form text fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

For more information about protecting sensitive information in AWS IoT Greengrass, see the section called "Don't log sensitive information".

For more information about data protection, see the AWS Shared Responsibility Model and GDPR blog post on the *AWS Security Blog*.

Topics

- Data encryption
- Hardware security integration

Data encryption

AWS IoT Greengrass uses encryption to protect data while in-transit (over the internet or local network) and at rest (stored in the AWS Cloud).

Devices in a AWS IoT Greengrass environment often collect data that's sent to AWS services for further processing. For more information about data encryption on other AWS services, see the security documentation for that service.

Topics

- Encryption in transit
- Encryption at rest
- Key management for the Greengrass core device

Encryption in transit

AWS IoT Greengrass has two modes of communication where data is in transit:

- the section called "Data in transit over the internet". Communication between a Greengrass core and AWS IoT Greengrass over the internet is encrypted.
- the section called "Data on the core device". Communication between components on the Greengrass core device is not encrypted.

Data in transit over the internet

AWS IoT Greengrass uses Transport Layer Security (TLS) to encrypt all communication over the internet. All data sent to the AWS Cloud is sent over a TLS connection using MQTT or HTTPS protocols, so it is secure by default. AWS IoT Greengrass uses the AWS IoT transport security model. For more information, see Transport security in the *AWS IoT Core Developer Guide*.

Data on the core device

AWS IoT Greengrass doesn't encrypt data exchanged locally on the Greengrass core device because the data doesn't leave the device. This includes communication between user-defined components, the AWS IoT device SDK, and public components, such as stream manager.

Encryption at rest

AWS IoT Greengrass stores your data:

- the section called “Data at rest in the AWS Cloud”. This data is encrypted.
- the section called “Data at rest on the Greengrass core”. This data is not encrypted (except local copies of your secrets).

Data at rest in the AWS Cloud

AWS IoT Greengrass encrypts customer data stored in the AWS Cloud. This data is protected using AWS KMS keys that are managed by AWS IoT Greengrass.

Data at rest on the Greengrass core

AWS IoT Greengrass relies on Unix file permissions and full-disk encryption (if enabled) to protect data at rest on the core. It is your responsibility to secure the file system and device.

However, AWS IoT Greengrass does encrypt local copies of your secrets retrieved from AWS Secrets Manager. For more information, see the secret manager component.

Key management for the Greengrass core device

It's the responsibility of the customer to guarantee secure storage of cryptographic (public and private) keys on the Greengrass core device. AWS IoT Greengrass uses public and private keys for the following scenario:

- The IoT client key is used with the IoT certificate to authenticate the Transport Layer Security (TLS) handshake when a Greengrass core connects to AWS IoT Core. For more information, see the section called “Device authentication and authorization”.

> **Note**
>
> The key and certificate are also referred to as the core private key and the core device certificate.

A Greengrass core device supports private key storage using file system permissions or a hardware security module. If you use file system-based private keys, you are responsible for their secure storage on the core device.

Hardware security integration

> **Note**
>
> For Greengrass nucleus, this feature requires v2.5.3 or later. AWS IoT Greengrass doesn't currently support this feature on Windows core devices.
> Greengrass nucleus lite v2.5.0 and later include built-in PKCS#11 support and do not require the PKCS#11 provider component. You must have an OpenSSL provider configured for your PKCS#11 device. For more information, see PKCS#11 support in the AWS IoT Greengrass nucleus lite GitHub repository.

You can configure the AWS IoT Greengrass Core software to use a hardware security module (HSM) through the PKCS#11 interface. This feature enables you to securely store the device's private key and certificate so that they aren't exposed or duplicated in software. You can store the private key and certificate on a hardware module such as an HSM or a Trusted Platform Module (TPM).

The AWS IoT Greengrass Core software uses a private key and X.509 certificate to authenticate connections to the AWS IoT and AWS IoT Greengrass services. The secret manager component uses this private key to securely encrypt and decrypt the secrets that you deploy to a Greengrass core device. When you configure a core device to use an HSM, these components use the private key and certificate that you store in the HSM.

The Moquette MQTT broker component also stores a private key for its local MQTT server certificate. This component store the private key on the device's file system in the component's work folder. Currently, AWS IoT Greengrass doesn't support storing this private key or certificate in an HSM.

> **Tip**
>
> Search for devices that support this feature in the AWS Partner Device Catalog.

Topics

- Requirements
- Hardware security best practices
- Install the Greengrass nucleus software with hardware security
- Configure hardware security on an existing Greengrass nucleus core device
- Use hardware without PKCS#11 support
- See also

Requirements

You must meet the following requirements to use an HSM on a Greengrass core device:

- Greengrass nucleus v2.5.3 or later, or Greengrass nucleus lite v2.5.0 or later, installed on the core device. If you use Greengrass nucleus lite, see PKCS#11 support in the AWS IoT Greengrass nucleus lite GitHub repository for setup instructions.
- The PKCS#11 provider component installed on the core device. You can download and install this component when you install the AWS IoT Greengrass Core software on a core device.
- A hardware security module that supports the PKCS#1 v1.5 signature scheme and RSA keys with an RSA-2048 key size (or larger) or ECC keys.

> **Note**
>
> To use a hardware security module with ECC keys, you must use Greengrass nucleus v2.5.6 or later.

To use a hardware security module and secret manager, you must use a hardware security module with RSA keys.

- A PKCS#11 provider library that the AWS IoT Greengrass Core software can load at runtime (using libdl) to invoke PKCS#11 functions. The PKCS#11 provider library must implement the following PKCS#11 API operations:
  - `C_Initialize`
  - `C_Finalize`
  - `C_GetSlotList`
  - `C_GetSlotInfo`
  - `C_GetTokenInfo`
  - `C_OpenSession`
  - `C_GetSessionInfo`
  - `C_CloseSession`
  - `C_Login`
  - `C_Logout`
  - `C_GetAttributeValue`
  - `C_FindObjectsInit`
  - `C_FindObjects`
  - `C_FindObjectsFinal`
  - `C_DecryptInit`
  - `C_Decrypt`
  - `C_DecryptUpdate`
  - `C_DecryptFinal`
  - `C_SignInit`
  - `C_Sign`
  - `C_SignUpdate`
  - `C_SignFinal`
  - `C_GetMechanismList`
  - `C_GetMechanismInfo`
  - `C_GetInfo`

  - `C_GetFunctionList`
- The hardware module must be resolvable by slot label, as defined in the PKCS#11 specification.
- You must store the private key and certificate in the HSM in the same slot, and they must use the same object label and object ID, if the HSM supports object IDs.
- The certificate and private key must be resolvable by object labels.
- The private key must have the following permissions:
  - `sign`
  - `decrypt`
- (Optional) To use the secret manager component, you must use version 2.1.0 or later, and the private key must have the following permissions:
  - `unwrap`
  - `wrap`

Hardware security best practices

Consider the following best practices when you configure hardware security on Greengrass core devices.

- Generate private keys directly on the HSM by using the internal hardware random-number generator. This approach is more secure than importing a private key that you generate elsewhere, because the private key remains within the HSM.
- Configure private keys to be immutable and prohibit export.
- Use the provisioning tool that the HSM hardware vendor recommends to generate a certifacte signing request (CSR) using the hardware-protected private key, and then use the AWS IoT console or API to generate a client certificate.

> **Note**
>
> The security best practice to rotate keys doesn't apply when you generate private keys on an HSM.

Install the Greengrass nucleus software with hardware security

When you install the AWS IoT Greengrass Core software, you can configure it to use a private key that you generate in an HSM. This approach follows the security best practice to generate the private key in the HSM, so the private key remains within the HSM.

To install the AWS IoT Greengrass Core software with hardware security, you do the following:

1. Generate a private key in the HSM.
2. Create a certificate signing request (CSR) from the private key.
3. Create a certificate from the CSR. You can create a certificate signed by AWS IoT or by another root certificate authority (CA). For more information about how to use another root CA, see Create your own client certificates in the *AWS IoT Core Developer Guide*.
4. Download the AWS IoT certificate and import it into the HSM.
5. Install the AWS IoT Greengrass Core software from a configuration file that specifies to use the PKCS#11 provider component and the private key and certificate in the HSM.

You can choose one of the following installation options to install the AWS IoT Greengrass Core software with hardware security:

- **Manual installation**

  Choose this option to manually create the required AWS resources and configure hardware security. For more information, see Install AWS IoT Greengrass Core software with manual resource provisioning.

- **Installation with custom provisioning**

  Choose this option to develop a custom Java application that automatically creates the required AWS resources and configures hardware security. For more information, see Install AWS IoT Greengrass Core software with custom resource provisioning.

The Greengrass nucleus doesn't support installing the AWS IoT Greengrass Core software with hardware security when you install with automatic resource provisioning or AWS IoT fleet provisioning. Greengrass nucleus lite v2.5.0 and later support hardware security with fleet provisioning.

Configure hardware security on an existing Greengrass nucleus core device

You can import a core device's private key and certificate to an HSM to configure hardware security.

Considerations

- You must have root access to the core device's file system.
- In this procedure, you shut down the AWS IoT Greengrass Core software, so the core device is offline and unavailable while you configure hardware security.

To configure hardware security on an existing core device, you do the following:

1. Initialize the HSM.
2. Deploy the PKCS#11 provider component to the core device.
3. Stop the AWS IoT Greengrass Core software.
4. Import the core device's private key and certificate to the HSM.
5. Update the AWS IoT Greengrass Core software's configuration file to use the private key and certificate in the HSM.
6. Start the AWS IoT Greengrass Core software.

Step 1: Initialize the hardware security module

Complete the following step to initialize the HSM on your core device.

To initialize the hardware security module

- Initialize an PKCS#11 token in the HSM, and save the slot ID and user PIN that for the token. Check the documentation for your HSM to learn how to initialize a token. You use the slot ID and user PIN later when you deploy and configure the PKCS#11 provider component.

Step 2: Deploy the PKCS#11 provider component

Complete the following steps to deploy and configure the PKCS#11 provider component. You can deploy the component to one or more core devices.

To deploy the PKCS#11 provider component (console)

1. In the AWS IoT Greengrass console navigation menu, choose **Components**.
2. On the **Components** page, choose the **Public components** tab, and then choose **aws.greengrass.crypto.Pkcs11Provider**.
3. On the **aws.greengrass.crypto.Pkcs11Provider** page, choose **Deploy**.
4. From **Add to deployment**, choose an existing deployment to revise, or choose to create a new deployment, and then choose **Next**.
5. If you chose to create a new deployment, choose the target core device or thing group for the deployment. On the **Specify target** page, under **Deployment target**, choose a core device or thing group, and then choose **Next**.
6. On the **Select components** page, under **Public components**, select **aws.greengrass.crypto.Pkcs11Provider**, and then choose **Next**.
7. On the **Configure components** page, select **aws.greengrass.crypto.Pkcs11Provider**, and then do the following:

   a. Choose **Configure component**.

   b. In the **Configure aws.greengrass.crypto.Pkcs11Provider** modal, under **Configuration update**, in **Configuration to merge**, enter the following configuration update. Update the following configuration parameters with values for the target core devices. Specify the slot ID and user PIN where you initialized the PKCS#11 token earlier. You import the private key and certificate into this slot in the HSM later.

      `name`

      A name for the PKCS#11 configuration.

      `library`

      The absolute file path to the PKCS#11 implementation's library that the AWS IoT Greengrass Core software can load with libdl.

      `slot`

      The ID of the slot that contains the private key and device certificate. This value is different than the slot index or slot label.

      `userPin`

      The user PIN to use to access the slot.

```
{
  "name": "softhsm_pkcs11",
  "library": "/usr/lib/softhsm/libsofthsm2.so",
  "slot": 1,
  "userPin": "1234"
}
```

c. Choose **Confirm** to close the modal, and then choose **Next**.

8. On the **Configure advanced settings** page, keep the default configuration settings, and choose **Next**.
9. On the **Review** page, choose **Deploy**.

   The deployment can take up to a minute to complete.

To deploy the PKCS#11 provider component (AWS CLI)

To deploy the PKCS#11 provider component, create a deployment document that includes `aws.greengrass.crypto.Pkcs11Provider` in the `components` object, and specify the configuration update for the component. Follow instructions in Create deployments to create a new deployment or revise an existing deployment.

The following example partial deployment document specifies to deploy and configure the PKCS#11 provider component. Update the following configuration parameters with values for the target core devices. Save the slot ID and user PIN to use later when you import the private key and certificate into the HSM.

`name`

A name for the PKCS#11 configuration.

`library`

The absolute file path to the PKCS#11 implementation's library that the AWS IoT Greengrass Core software can load with libdl.

`slot`

The ID of the slot that contains the private key and device certificate. This value is different than the slot index or slot label.

`userPin`

The user PIN to use to access the slot.

```
{
  "name": "softhsm_pkcs11",
  "library": "/usr/lib/softhsm/libsofthsm2.so",
  "slot": 1,
  "userPin": "1234"
}
```

```
{
  ...,
  "components": {
    ...,
    "aws.greengrass.crypto.Pkcs11Provider": {
      "componentVersion": "2.0.0",
      "configurationUpdate": {
        "merge": "{\"name\":\"softhsm_pkcs11\",\"library\":\"/usr/lib/softhsm/
libsofthsm2.so\",\"slot\":1,\"userPin\":\"1234\"}"
      }
    }
  }
}
```

The deployment can take several minutes to complete. You can use the AWS IoT Greengrass service to check the status of the deployment. You can check the AWS IoT Greengrass Core software logs to verify that the PKCS#11 provider component deploys successfully. For more information, see the following:

- Check deployment status
- Monitor AWS IoT Greengrass logs

If the deployment fails, you can troubleshoot the deployment on each core device. For more information, see Troubleshooting AWS IoT Greengrass V2.

Step 3: Update the configuration on the core device

The AWS IoT Greengrass Core software uses a configuration file that specifies how the device operates. This configuration file includes where to find the private key and certificate that the

device uses to connect to the AWS Cloud. Complete the following steps to import the core device's private key and certificate into the HSM and update the configuration file to use the HSM.

To update the configuration on the core device to use hardware security

1. Stop the AWS IoT Greengrass Core software. If you configured the AWS IoT Greengrass Core software as a system service with systemd, you can run the following command to stop the software.

   ```
   sudo systemctl stop greengrass.service
   ```

2. Find the core device's private key and certificate files.

   - If you installed the AWS IoT Greengrass Core software with automatic provisioning or fleet provisioning, the private key exists at */greengrass/v2*`/privKey.key`, and the certificate exists at */greengrass/v2*`/thingCert.crt`.
   - If you installed the AWS IoT Greengrass Core software with manual provisioning, the private key exists at */greengrass/v2*`/private.pem.key` by default, and the certificate exists at */greengrass/v2*`/device.pem.crt` by default.

   You can also check the `system.privateKeyPath` and `system.certificateFilePath` properties in */greengrass/v2*`/config/effectiveConfig.yaml` to find the location of these files.

3. Import the private key and certificate into the HSM. Check the documentation for your HSM to learn how to import private keys and certificates into it. Import the private key and certificate using the slot ID and user PIN where you initialized the PKCS#11 token earlier. You must use the same object label and object ID for the private key and the certificate. Save the object label that you specify when you import each file. You use this label later when you update the AWS IoT Greengrass Core software configuration to use the private key and certificate in the HSM.

4. Update the AWS IoT Greengrass Core configuration to use the private key and certificate in the HSM. To update the configuration, you modify the AWS IoT Greengrass Core configuration file and run the AWS IoT Greengrass Core software with the updated configuration file to apply the new configuration.

   Do the following:

a. Create a back up of the AWS IoT Greengrass Core configuration file. You can use this back up to restore the core device if you run into issues when you configure hardware security.

```
sudo cp /greengrass/v2/config/effectiveConfig.yaml ~/ggc-config-backup.yaml
```

b. Open the AWS IoT Greengrass Core configuration file in a text editor. For example, you can run the following command to use GNU nano to edit the file. Replace */greengrass/v2* with the path to the Greengrass root folder.

```
sudo nano /greengrass/v2/config/effectiveConfig.yaml
```

c. Replace the value of the `system.privateKeyPath` with the PKCS#11 URI for the private key in the HSM. Replace *iotdevicekey* with the object label where you imported the private key and certificate earlier.

```
pkcs11:object=iotdevicekey;type=private
```

d. Replace the value of the `system.certificateFilePath` with the PKCS#11 URI for the certificate in the HSM. Replace *iotdevicekey* with the object label where you imported the private key and certificate earlier.

```
pkcs11:object=iotdevicekey;type=cert
```

After you finish these steps, the `system` property in the AWS IoT Greengrass Core configuration file should look similar to the following example.

```
system:
  certificateFilePath: "pkcs11:object=iotdevicekey;type=cert"
  privateKeyPath: "pkcs11:object=iotdevicekey;type=private"
  rootCaPath: "/greengrass/v2/rootCA.pem"
  rootpath: "/greengrass/v2"
  thingName: "MyGreengrassCore"
```

5. Apply the configuration in the updated `effectiveConfig.yaml` file. Run `Greengrass.jar` with the `--init-config` parameter to apply the configuration in `effectiveConfig.yaml`. Replace */greengrass/v2* with the path to the Greengrass root folder.

```
sudo java -Droot="/greengrass/v2" \
```

```
  -jar /greengrass/v2/alts/current/distro/lib/Greengrass.jar \
  --start false \
  --init-config /greengrass/v2/config/effectiveConfig.yaml
```

6. Start the AWS IoT Greengrass Core software. If you configured the AWS IoT Greengrass Core software as a system service with systemd, you can run the following command to start the software.

   ```
   sudo systemctl start greengrass.service
   ```

   For more information, see Run the AWS IoT Greengrass Core software.

7. Check the AWS IoT Greengrass Core software logs to verify that the software starts and connects to the AWS Cloud. The AWS IoT Greengrass Core software uses the private key and certificate to connect to the AWS IoT and AWS IoT Greengrass services.

   ```
   sudo tail -f /greengrass/v2/logs/greengrass.log
   ```

   The following INFO-level log messages indicate that the AWS IoT Greengrass Core software successfully connects to the AWS IoT and AWS IoT Greengrass services.

   ```
   2021-12-06T22:47:53.702Z [INFO] (Thread-3)
    com.aws.greengrass.mqttclient.AwsIotMqttClient: Successfully connected to AWS IoT
    Core. {clientId=MyGreengrassCore5, sessionPresent=false}
   ```

8. (Optional) After you verify that the AWS IoT Greengrass Core software works with the private key and certificate in the HSM, delete the private key and certificate files from the device's file system. Run the following command, and replace the file paths with the paths to the private key and certificate files.

   ```
   sudo rm /greengrass/v2/privKey.key
   sudo rm /greengrass/v2/thingCert.crt
   ```

Use hardware without PKCS#11 support

The PKCS#11 library is typically provided by the hardware vendor or is open source. For example, with standards-compliant hardware (such as TPM1.2), it might be possible to use existing open source software. However, if your hardware doesn't have a corresponding PKCS#11 library

implementation, or if you want to write a custom PKCS#11 provider, contact your Amazon Web Services Enterprise Support representative with integration-related questions.

See also

- PKCS #11 Cryptographic Token Interface Usage Guide Version 2.4.0
- RFC 7512
- PKCS #1: RSA Encryption Version 1.5

Device authentication and authorization for AWS IoT Greengrass

Devices in AWS IoT Greengrass environments use X.509 certificates for authentication and AWS IoT policies for authorization. Certificates and policies allow devices to securely connect with each other, AWS IoT Core, and AWS IoT Greengrass.

X.509 certificates are digital certificates that use the X.509 public key infrastructure standard to associate a public key with the identity contained in a certificate. X.509 certificates are issued by a trusted entity called a certificate authority (CA). The CA maintains one or more special certificates called CA certificates that it uses to issue X.509 certificates. Only the certificate authority has access to CA certificates.

AWS IoT policies define the set of operations allowed for AWS IoT devices. Specifically, they allow and deny access to AWS IoT Core and AWS IoT Greengrass data plane operations, such as publishing MQTT messages and retrieving device shadows.

All devices require an entry in the AWS IoT Core registry and an activated X.509 certificate with an attached AWS IoT policy. Devices fall into two categories:

- **Greengrass core devices**

  Greengrass core devices use certificates and AWS IoT policies to connect to AWS IoT Core and AWS IoT Greengrass. The certificates and policies also allow AWS IoT Greengrass to deploy components and configurations to core devices.
- **Client devices**

  MQTT client devices use certificates and policies to connect to AWS IoT Core and the AWS IoT Greengrass service. This enables client devices to use the AWS IoT Greengrass cloud discovery to

find and connect to a Greengrass core device. A client device uses the same certificate to connect to the AWS IoT Core cloud service and core devices. Client devices also use discovery information for mutual authentication with the core device. For more information, see Interact with local IoT devices.

X.509 certificates

Communication between core devices and client devices and between devices and AWS IoT Core or AWS IoT Greengrass must be authenticated. This mutual authentication is based on registered X.509 device certificates and cryptographic keys.

In an AWS IoT Greengrass environment, devices use certificates with public and private keys for the following Transport Layer Security (TLS) connections:

- The AWS IoT client component on the Greengrass core device that connects to AWS IoT Core and AWS IoT Greengrass over the internet.
- Client devices that connect to AWS IoT Greengrass over the internet to discover core devices.
- The MQTT broker component on the Greengrass core connecting to Greengrass devices in the group over the local network.

AWS IoT Greengrass core devices store certificates in the Greengrass root folder.

Certificate authority (CA) certificates

Greengrass core devices and client devices download a root CA certificate used for authentication with the AWS IoT Core and AWS IoT Greengrass services. We recommend that you use an Amazon Trust Services (ATS) root CA certificate, such as Amazon Root CA 1. For more information, see CA certificates for server authentication in the *AWS IoT Core Developer Guide*.

Client devices also download a Greengrass core device CA certificate. They use this certificate to validate the MQTT server certificate on the core device during mutual authentication.

Certificate rotation on the local MQTT broker

When you enable client device support, Greengrass core devices generate a local MQTT server certificate that client devices use for mutual authentication. This certificate is signed by the core device CA certificate, which the core device stores in the AWS IoT Greengrass cloud. Client devices

retrieve the core device CA certificate when they discover the core device. They use the core device CA certificate to verify the core device's MQTT server certificate when they connect to the core device. The core device CA certificate expires after 5 years.

The MQTT server certificate expires every 7 days by default, and you can configure this duration to between 2 and 10 days. This limited period is based on security best practices. This rotation helps mitigate the threat of an attacker stealing the MQTT server certificate and private key to impersonate the Greengrass core device.

The Greengrass core device rotates the MQTT server certificate 24 hours before it expires. The Greengrass core device generates a new certificate and restarts the local MQTT broker. When this happens, all client devices connected to the Greengrass core device are disconnected. Client devices can reconnect to the Greengrass core device after a short period of time.

AWS IoT policies for data plane operations

Use AWS IoT policies to authorize access to the AWS IoT Core and AWS IoT Greengrass data planes. The AWS IoT Core data plane provides operations for devices, users, and applications. These operations include the ability to connect to AWS IoT Core and subscribe to topics. The AWS IoT Greengrass data plane provides operations for Greengrass devices. For more information, see AWS IoT Greengrass V2 policy actions. These operations include the ability to resolve component dependencies and download public component artifacts.

An AWS IoT policy is a JSON document that's similar to an IAM policy. It contains one or more policy statements that specify the following properties:

- `Effect`. The access mode, which can be `Allow` or `Deny`.
- `Action`. The list of actions that are allowed or denied by the policy.
- `Resource`. The list of resources on which the action is allowed or denied.

AWS IoT policies support * as a wildcard character, and treat MQTT wildcard characters (+ and #) as literal strings. For more information about the * wildcard, see Using wildcard in resource ARNs in the *AWS Identity and Access Management User Guide*.

For more information, see AWS IoT policies and AWS IoT policy actions in the *AWS IoT Core Developer Guide*.

Important

Thing policy variables (`iot:Connection.Thing.*`) aren't supported for in AWS IoT policies for core devices or Greengrass data plane operations. Instead, you can use a wildcard that matches multiple devices that have similar names. For example, you can specify `MyGreengrassDevice*` to match `MyGreengrassDevice1`, `MyGreengrassDevice2`, and so on.

Note

AWS IoT Core enables you to attach AWS IoT policies to thing groups to define permissions for groups of devices. Thing group policies don't allow access to AWS IoT Greengrass data plane operations. To allow a thing access to an AWS IoT Greengrass data plane operation, add the permission to an AWS IoT policy that you attach to the thing's certificate.

AWS IoT Greengrass V2 policy actions

AWS IoT Greengrass V2 defines the following policy actions that Greengrass core devices and client devices can use in AWS IoT policies. To specify a resource for an policy action, you use the Amazon Resource Name (ARN) of the resource.

Core device actions

`greengrass:GetComponentVersionArtifact`

Grants permission to get a presigned URL to download a public component artifact or a Lambda component artifact.

This permission is evaluated when a core device receives a deployment that specifies a public component or a Lambda that has artifacts. If the core device already has the artifact, it doesn't download the artifact again.

Resource type: `componentVersion`

Resource ARN format: `arn:aws:greengrass:`*`region`*`:`*`account-id`*`:components:`*`component-name`*`:versions:`*`component-version`*

`greengrass:ResolveComponentCandidates`

Grants permission to identify a list of components that meet the component, version, and platform requirements for a deployment. If the requirements conflict, or no components exist that meet the requirements, this operation returns an error and the deployment fails on the device.

This permission is evaluated when a core device receives a deployment that specifies components.

Resource type: None

Resource ARN format: *

`greengrass:GetDeploymentConfiguration`

Grants permission to get a presigned URL to download a large deployment document.

This permission is evaluated when a core device receives a deployment that specifies a deployment document larger than 7 KB (if the deployment targets a thing) or 31 KB (if the deployment targets a thing group). The deployment document includes component configurations, deployment policies, and deployment metadata. For more information, see Deploy AWS IoT Greengrass components to devices.

This feature is available for v2.3.0 and later of the Greengrass nucleus component.

Resource type: None

Resource ARN format: *

`greengrass:ListThingGroupsForCoreDevice`

Grants permission to get a core device's thing group hierarchy.

This permission is checked when a core device receives a deployment from AWS IoT Greengrass. The core device uses this action to identify whether it was removed from a thing group since the last deployment. If the core device was removed from a thing group, and that thing group is the target of a deployment to the core device, then the core device removes the components installed by that deployment.

This feature is used by v2.5.0 and later of the Greengrass nucleus component.

Resource type: `thing` (core device)

Resource ARN format: `arn:aws:iot:region:account-id:thing/core-device-thing-name`

`greengrass:VerifyClientDeviceIdentity`

Grants permission to verify the identity of a client device that connects to a core device.

This permission is evaluated when a core device runs the client device auth component and receives an MQTT connection from a client device. The client device presents its AWS IoT device certificate. Then, the core device sends the device certificate to the AWS IoT Greengrass cloud service to verify the client device's identity. For more information, see Interact with local IoT devices.

Resource type: None

Resource ARN format: *

`greengrass:VerifyClientDeviceIoTCertificateAssociation`

Grants permission to verify whether a client device is associated with an AWS IoT certificate.

This permission is evaluated when a core device runs the client device auth component and authorizes a client device to connect over MQTT. For more information, see Interact with local IoT devices.

> **Note**
>
> For a core device to use this operation, the Greengrass service role must be associated to your AWS account and allow the `iot:DescribeCertificate` permission.

Resource type: `thing` (client device)

Resource ARN format: `arn:aws:iot:region:account-id:thing/client-device-thing-name`

`greengrass:PutCertificateAuthorities`

Grants permission to upload certificate authority (CA) certificates that client devices can download to verify the core device.

This permission is evaluated when a core device installs and runs the client device auth component. This component creates a local certificate authority and uses this operation to upload its CA certificates. Client devices download these CA certificates when they use the

Discover operation to find core devices where they can connect. When client devices connect to an MQTT broker on a core device, they use these CA certificates to verify the identity of the core device. For more information, see Interact with local IoT devices.

Resource type: None

ARN format: *

`greengrass:GetConnectivityInfo`

Grants permission to get connectivity information for a core device. This information describes how client devices can connect to the core device.

This permission is evaluated when a core device installs and runs the client device auth component. This component uses the connectivity information to generate valid CA certificates to upload to the AWS IoT Greengrass cloud service with the PutCertificateAuthories operation. Client devices use these CA certificates to verify the identity of the core device. For more information, see Interact with local IoT devices.

You can also use this operation on the AWS IoT Greengrass control plane to view connectivity information for a core device. For more information, see GetConnectivityInfo in the *AWS IoT Greengrass V1 API Reference*.

Resource type: `thing` (core device)

Resource ARN format: `arn:aws:iot:`*`region`*`:`*`account-id`*`:thing/`*`core-device-thing-name`*

`greengrass:UpdateConnectivityInfo`

Grants permission to update connectivity information for a core device. This information describes how client devices can connect to the core device.

This permission is evaluated when a core device runs the IP detector component. This component identifies the information that client devices require to connect to the core device on the local network. Then, this component uses this operation to upload the connectivity information to the AWS IoT Greengrass cloud service, so client devices can retrieve this information with the Discover operation. For more information, see Interact with local IoT devices.

You can also use this operation on the AWS IoT Greengrass control plane to manually update connectivity information for a core device. For more information, see UpdateConnectivityInfo in the *AWS IoT Greengrass V1 API Reference*.

Resource type: `thing` (core device)

Resource ARN format: `arn:aws:iot:`*`region`*`:`*`account-id`*`:thing/`*`core-device-thing-name`*

Client device actions

`greengrass:Discover`

Grants permission to discover connectivity information for core devices where a client device can connect. This information describes how the client device can connect to the core devices. A client device can discover only the core devices that you have associated it with by using the BatchAssociateClientDeviceWithCoreDevice operation. For more information, see Interact with local IoT devices.

Resource type: `thing` (client device)

Resource ARN format: `arn:aws:iot:`*`region`*`:`*`account-id`*`:thing/`*`client-device-thing-name`*

Update a core device's AWS IoT policy

You can use the AWS IoT Greengrass and AWS IoT consoles or the AWS IoT API to view and update a core device's AWS IoT policy.

> **Note**
>
> If you used the AWS IoT Greengrass Core software installer to provision resources, your core device has an AWS IoT policy that allows access to all AWS IoT Greengrass actions (`greengrass:*`). You can follow these steps to restrict access to only the actions that a core device uses.

Review and update a core device's AWS IoT policy (console)

1. In the AWS IoT Greengrass console navigation menu, choose **Core devices**.
2. On the **Core devices** page, choose the core device to update.
3. On the core device details page, choose the link to the core device's **Thing**. This link opens the thing details page in the AWS IoT console.

4. On the thing details page, choose **Certificates**.
5. In the **Certificates** tab, choose the thing's active certificate.
6. On the certificate details page, choose **Policies**.
7. In the **Policies** tab, choose the AWS IoT policy to review and update. You can add the required permissions to any policy that is attached to the core device's active certificate.

   > **Note**
   >
   > If you used the AWS IoT Greengrass Core software installer to provision resources, you have two AWS IoT policies. We recommend that you choose the policy named **GreengrassV2IoTThingPolicy**, if it exists. Core devices that you create with the quick installer use this policy name by default. If you add permissions to this policy, you are also granting these permissions to other core devices that use this policy.

8. In the policy overview, choose **Edit active version**.
9. Review the policy and add, remove, or edit permissions as needed.
10. To set a new policy version as the active version, under **Policy version status**, select **Set the edited version as the active version for this policy**.
11. Choose **Save as new version**.

Review and update a core device's AWS IoT policy (AWS CLI)

1. List the principals for the core device's AWS IoT thing. Thing principals can be X.509 device certificates or other identifies. Run the following command, and replace *MyGreengrassCore* with the name of the core device.

   ```
   aws iot list-thing-principals --thing-name MyGreengrassCore
   ```

   The operation returns a response that lists the core device's thing principals.

   ```
   {
       "principals": [
           "arn:aws:iot:us-west-2:123456789012:cert/certificateId"
       ]
   }
   ```

2. Identify the core device's active certificate. Run the following command, and replace *certificateId* with the ID of each certificate from the previous step until you find the active certificate. The certificate ID is the hexadecimal string at the end of the certificate ARN. The `--query` argument specifies to output only the certificate's status.

   ```
   aws iot describe-certificate --certificate-id certificateId --query
    'certificateDescription.status'
   ```

   The operation returns the certificate status as a string. For example, if the certificate is active, this operation outputs `"ACTIVE"`.

3. List the AWS IoT policies that are attached to the certificate. Run the following command, and replace the certificate ARN with the ARN of the certificate.

   ```
   aws iot list-principal-policies --principal arn:aws:iot:us-
   west-2:123456789012:cert/certificateId
   ```

   The operation returns a response that lists the AWS IoT policies that are attached to the certificate.

   ```
   {
       "policies": [
           {
               "policyName":
    "GreengrassTESCertificatePolicyMyGreengrassCoreTokenExchangeRoleAlias",
               "policyArn": "arn:aws:iot:us-west-2:123456789012:policy/
   GreengrassTESCertificatePolicyMyGreengrassCoreTokenExchangeRoleAlias"
           },
           {
               "policyName": "GreengrassV2IoTThingPolicy",
               "policyArn": "arn:aws:iot:us-west-2:123456789012:policy/
   GreengrassV2IoTThingPolicy"
           }
       ]
   }
   ```

4. Choose the policy to view and update.

> **Note**
>
> If you used the AWS IoT Greengrass Core software installer to provision resources, you have two AWS IoT policies. We recommend that you choose the policy named **GreengrassV2IoTThingPolicy**, if it exists. Core devices that you create with the quick installer use this policy name by default. If you add permissions to this policy, you are also granting these permissions to other core devices that use this policy.

5. Get the policy's document. Run the following command, and replace *GreengrassV2IoTThingPolicy* with the name of the policy.

```
aws iot get-policy --policy-name GreengrassV2IoTThingPolicy
```

The operation returns a response that contains the policy's document and other information about the policy. The policy document is a JSON object serialized as a string.

```
{
    "policyName": "GreengrassV2IoTThingPolicy",
    "policyArn": "arn:aws:iot:us-west-2:123456789012:policy/
GreengrassV2IoTThingPolicy",
    "policyDocument": "{\
  \\"Version\\": \\"2012-10-17       \\",\
  \\"Statement\\": [\
    {\
      \\"Effect\\": \\"Allow\\",\
      \\"Action\\": [\
                \\"iot:Connect\\",\
                \\"iot:Publish\\",\
                \\"iot:Subscribe\\",\
                \\"iot:Receive\\",\
                \\"greengrass:*\\"\
],\
      \\"Resource\\": \\"*\\"\
    }\
  ]\
}",
    "defaultVersionId": "1",
    "creationDate": "2021-02-05T16:03:14.098000-08:00",
    "lastModifiedDate": "2021-02-05T16:03:14.098000-08:00",
```

```
    "generationId":
 "f19144b798534f52c619d44f771a354f1b957dfa2b850625d9f1d0fde530e75f"
}
```

6. Use an online converter or other tool to convert the policy document string to a JSON object, and then save it to a file named `iot-policy.json`.

   For example, if you have the jq tool installed, you can run the following command to get the policy document, convert it to a JSON object, and save the policy document as a JSON object.

   ```
   aws iot get-policy --policy-name GreengrassV2IoTThingPolicy --query
    'policyDocument' | jq fromjson >> iot-policy.json
   ```

7. Review the policy document, and add, remove, or edit permissions as needed.

   For example, on a Linux-based system, you can run the following command to use GNU nano to open the file.

   ```
   nano iot-policy.json
   ```

   When you're done, the policy document might look similar to the minimal AWS IoT policy for core devices.

8. Save the changes as a new version of the policy. Run the following command, and replace *GreengrassV2IoTThingPolicy* with the name of the policy.

   ```
   aws iot create-policy-version --policy-name GreengrassV2IoTThingPolicy --policy-
   document file://iot-policy.json --set-as-default
   ```

   The operation returns a response similar to the following example if it succeeds.

   ```
   {
       "policyArn": "arn:aws:iot:us-west-2:123456789012:policy/
   GreengrassV2IoTThingPolicy",
       "policyDocument": "{\
     \\"Version\\": \\"2012-10-17       \\",\
     \\"Statement\\": [\
       {\
         \\"Effect\\": \\"Allow\\",\
         \\"Action\\": [\
   \\t\\t\\"iot:Connect\\",\
   ```

```
\\t\\t\\"iot:Publish\\",\
\\t\\t\\"iot:Subscribe\\",\
\\t\\t\\"iot:Receive\\",\
\\t\\t\\"greengrass:*\\"\
      ],\
      \\"Resource\\": \\"*\\"\
    }\
  ]\
}",
    "policyVersionId": "2",
    "isDefaultVersion": true
}
```

Minimal AWS IoT policy for AWS IoT Greengrass V2 core devices

Important

Later versions of the Greengrass nucleus component require additional permissions on the minimal AWS IoT policy. You might need to update your core devices' AWS IoT policies to grant additional permissions.

- Core devices that run Greengrass nucleus v2.5.0 and later use the `greengrass:ListThingGroupsForCoreDevice` permission to uninstall components when you remove a core device from a thing group.
- Core devices that run Greengrass nucleus v2.3.0 and later use the `greengrass:GetDeploymentConfiguration` permission to support large deployment configuration documents.

The following example policy includes the minimum set of actions required to support basic Greengrass functionality for your core device.

- The `Connect` policy includes the * wildcard after the core device thing name (for example, *`core-device-thing-name`**). The core device uses the same device certificate to make multiple concurrent subscriptions to AWS IoT Core, but the client ID in a connection might not be an exact match of the core device thing name. After the first 50 subscriptions, the core device uses *`core-device-thing-name`*#*`number`* as the client ID, where *`number`* increments for each

additional 50 subscriptions. For example, when a core device named `MyCoreDevice` creates 150 concurrent subscriptions, it uses the following client IDs:

- Subscriptions 1 to 50: `MyCoreDevice`
- Subscriptions 51 to 100: `MyCoreDevice#2`
- Subscriptions 101 to 150: `MyCoreDevice#3`

The wildcard allows the core device to connect when it uses these client IDs that have a suffix.

- The policy lists the MQTT topics and topic filters that the core device can publish messages to, subscribe to, and receive messages on, including topics used for shadow state. To support message exchange between AWS IoT Core, Greengrass components, and client devices, specify the topics and topic filters that you want to allow. For more information, see Publish/Subscribe policy examples in the *AWS IoT Core Developer Guide*.
- The policy grants permission to publish to the following topic for telemetry data.

  ```
  $aws/things/core-device-thing-name/greengrass/health/json
  ```

  You can remove this permission for core devices where you disable telemetry. For more information, see Gather system health telemetry data from AWS IoT Greengrass core devices.
- The policy grants permission to assume an IAM role through an AWS IoT role alias. The core device uses this role, called the token exchange role, to acquire AWS credentials that it can use to authenticate AWS requests. For more information, see Authorize core devices to interact with AWS services.

  When you install the AWS IoT Greengrass Core software, you create and attach a second AWS IoT policy that includes only this permission. If you include this permission in your core device's primary AWS IoT policy, you can detach and delete the other AWS IoT policy.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
```

```
            ],
            "Resource": "arn:aws:iot:us-east-1:123456789012:client/core-device-
thing-name*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Receive",
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/things/core-
device-thing-name/greengrass/health/json",
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/things/core-
device-thing-name/greengrassv2/health/json",
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/things/core-
device-thing-name/jobs/*",
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/things/core-
device-thing-name/shadow/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/things/core-
device-thing-name/jobs/*",
                "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/things/core-
device-thing-name/shadow/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": "iot:AssumeRoleWithCertificate",
            "Resource": "arn:aws:iot:us-east-1:123456789012:rolealias/token-
exchange-role-alias-name"
        },
        {
            "Effect": "Allow",
            "Action": [
                "greengrass:GetComponentVersionArtifact",
                "greengrass:ResolveComponentCandidates",
```

```
                "greengrass:GetDeploymentConfiguration",
                "greengrass:ListThingGroupsForCoreDevice"
            ],
            "Resource": "*"
        }
    ]
}
```

Minimal AWS IoT policy to support client devices

The following example policy includes the minimum set of actions required to support interaction with client devices on a core device. To support client devices, a core device must have the permissions in this AWS IoT policy in addition to the Minimal AWS IoT policy for basic operation.

- The policy allows the core device to update its own connectivity information. This permission (`greengrass:UpdateConnectivityInfo`) is required only if you deploy the IP detector component to the core device.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/things/core-device-thing-name-gci/shadow/get"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
```

```
                "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/things/core-
device-thing-name-gci/shadow/update/delta",
                "arn:aws:iot:us-east-1:123456789012:topicfilter/$aws/things/core-
device-thing-name-gci/shadow/get/accepted"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Receive"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/things/core-
device-thing-name-gci/shadow/update/delta",
                "arn:aws:iot:us-east-1:123456789012:topic/$aws/things/core-
device-thing-name-gci/shadow/get/accepted"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "greengrass:PutCertificateAuthorities",
                "greengrass:VerifyClientDeviceIdentity"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "greengrass:VerifyClientDeviceIoTCertificateAssociation"
            ],
            "Resource": "arn:aws:iot:us-east-1:123456789012:thing/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "greengrass:GetConnectivityInfo",
                "greengrass:UpdateConnectivityInfo"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:thing/core-device-thing-name"
            ]
        }
    ]
```

```
}
```

Minimal AWS IoT policy for client devices

The following example policy includes the minimum set of actions required for a client device to discover core devices where they connect and communicate over MQTT. The client device's AWS IoT policy must include the `greengrass:Discover` action to allow the device to discover connectivity information for its associated Greengrass core devices. In the `Resource` section, specify the Amazon Resource Name (ARN) of the client device, not the ARN of the Greengrass core device.

- The policy allows communication on all MQTT topics. To follow best security practices, restrict the `iot:Publish`, `iot:Subscribe`, and `iot:Receive` permissions to the minimal set of topics that a client device requires for your use case.
- The policy allows the thing to discover core devices for all AWS IoT things. To follow best security practices, restrict the `greengrass:Discover` permission to the client device's AWS IoT thing or a wildcard that matches a set of AWS IoT things.

  > **Important**
  >
  > Thing policy variables (`iot:Connection.Thing.*`) aren't supported for in AWS IoT policies for core devices or Greengrass data plane operations. Instead, you can use a wildcard that matches multiple devices that have similar names. For example, you can specify `MyGreengrassDevice*` to match `MyGreengrassDevice1`, `MyGreengrassDevice2`, and so on.

- A client device's AWS IoT policy doesn't typically require permissions for `iot:GetThingShadow`, `iot:UpdateThingShadow`, or `iot:DeleteThingShadow` actions, because the Greengrass core device handles shadow sync operations for client devices. To enable the core device to handle client device shadows, check that the core device's AWS IoT policy allows these actions, and that the `Resource` section includes the ARNs of the client devices.

JSON

```
{
    "Version":"2012-10-17",
```

```
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot:Connect"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Publish"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Subscribe"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topicfilter/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "iot:Receive"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:topic/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "greengrass:Discover"
            ],
            "Resource": [
                "arn:aws:iot:us-east-1:123456789012:thing/*"
            ]
        }
```

```
    ]
}
```

Identity and access management for AWS IoT Greengrass

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use AWS IoT Greengrass resources. IAM is an AWS service that you can use with no additional charge.

> **Note**
>
> This topic describes IAM concepts and features. For information about IAM features supported by AWS IoT Greengrass, see the section called "How AWS IoT Greengrass works with IAM".

Audience

How you use AWS Identity and Access Management (IAM) differs based on your role:

- **Service user** - request permissions from your administrator if you cannot access features (see Troubleshooting identity and access issues for AWS IoT Greengrass)
- **Service administrator** - determine user access and submit permission requests (see How AWS IoT Greengrass works with IAM)
- **IAM administrator** - write policies to manage access (see Identity-based policy examples for AWS IoT Greengrass)

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be authenticated as the AWS account root user, an IAM user, or by assuming an IAM role.

You can sign in as a federated identity using credentials from an identity source like AWS IAM Identity Center (IAM Identity Center), single sign-on authentication, or Google/Facebook credentials. For more information about signing in, see How to sign in to your AWS account in the *AWS Sign-In User Guide*.

For programmatic access, AWS provides an SDK and CLI to cryptographically sign requests. For more information, see AWS Signature Version 4 for API requests in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity called the AWS account *root user* that has complete access to all AWS services and resources. We strongly recommend that you don't use the root user for everyday tasks. For tasks that require root user credentials, see Tasks that require root user credentials in the *IAM User Guide*.

IAM users and groups

An *IAM user* is an identity with specific permissions for a single person or application. We recommend using temporary credentials instead of IAM users with long-term credentials. For more information, see Require human users to use federation with an identity provider to access AWS using temporary credentials in the *IAM User Guide*.

An *IAM group* specifies a collection of IAM users and makes permissions easier to manage for large sets of users. For more information, see Use cases for IAM users in the *IAM User Guide*.

IAM roles

An *IAM role* is an identity with specific permissions that provides temporary credentials. You can assume a role by switching from a user to an IAM role (console) or by calling an AWS CLI or AWS API operation. For more information, see Methods to assume a role in the *IAM User Guide*.

IAM roles are useful for federated user access, temporary IAM user permissions, cross-account access, cross-service access, and applications running on Amazon EC2. For more information, see Cross account resource access in IAM in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy defines permissions when associated with an identity or resource. AWS evaluates these policies when a principal makes a request. Most policies are stored in AWS as JSON documents. For more information about JSON policy documents, see Overview of JSON policies in the *IAM User Guide*.

Using policies, administrators specify who has access to what by defining which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. An IAM administrator creates IAM policies and adds them to roles, which users can then assume. IAM policies define permissions regardless of the method used to perform the operation.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you attach to an identity (user, group, or role). These policies control what actions identities can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

Identity-based policies can be *inline policies* (embedded directly into a single identity) or *managed policies* (standalone policies attached to multiple identities). To learn how to choose between managed and inline policies, see Choose between managed policies and inline policies in the *IAM User Guide*.

Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples include IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. You must specify a principal in a resource-based policy.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Access control lists (ACLs)

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see Access control list (ACL) overview in the *Amazon Simple Storage Service Developer Guide*.

Other policy types

AWS supports additional policy types that can set the maximum permissions granted by more common policy types:

- **Permissions boundaries** – Set the maximum permissions that an identity-based policy can grant to an IAM entity. For more information, see Permissions boundaries for IAM entities in the *IAM User Guide*.
- **Service control policies (SCPs)** – Specify the maximum permissions for an organization or organizational unit in AWS Organizations. For more information, see Service control policies in the *AWS Organizations User Guide*.
- **Resource control policies (RCPs)** – Set the maximum available permissions for resources in your accounts. For more information, see Resource control policies (RCPs) in the *AWS Organizations User Guide*.
- **Session policies** – Advanced policies passed as a parameter when creating a temporary session for a role or federated user. For more information, see Session policies in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see Policy evaluation logic in the *IAM User Guide*.

See also

- the section called “How AWS IoT Greengrass works with IAM”
- the section called “Identity-based policy examples”
- the section called “Troubleshooting identity and access issues”

How AWS IoT Greengrass works with IAM

Before you use IAM to manage access to AWS IoT Greengrass, you should understand the IAM features that you can use with AWS IoT Greengrass.

IAM feature	Supported by Greengrass?
Identity-based policies with resource-level permissions	Yes
Resource-based policies	No

IAM feature	Supported by Greengrass?
Access control lists (ACLs)	No
Tags-based authorization	Yes
Temporary credentials	Yes
Service-linked roles	No
Service roles	Yes

For a high-level view of how other AWS services work with IAM, see AWS services that work with IAM in the *IAM User Guide*.

Identity-based policies for AWS IoT Greengrass

With IAM identity-based policies, you can specify allowed or denied actions and resources and the conditions under which actions are allowed or denied. AWS IoT Greengrass supports specific actions, resources, and condition keys. To learn about all of the elements that you use in a policy, see IAM JSON policy elements reference in the *IAM User Guide*.

Actions

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Action` element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Include actions in a policy to grant permissions to perform the associated operation.

Policy actions for AWS IoT Greengrass use the `greengrass:` prefix before the action. For example, to allow someone to use the `ListCoreDevices` API operation to list the core devices in their AWS account, you include the `greengrass:ListCoreDevices` action in their policy. Policy statements must include either an `Action` or `NotAction` element. AWS IoT Greengrass defines its own set of actions that describe tasks that you can perform with this service.

To specify multiple actions in a single statement, list them between brackets (`[ ]`) and separate them with commas, as follows:

```
"Action": [
  "greengrass:action1",
  "greengrass:action2",
  "greengrass:action3"
]
```

You can use wildcards (*) to specify multiple actions. For example, to specify all actions that begin with the word `List`, include the following action:

```
"Action": "greengrass:List*"
```

> **Note**
>
> We recommend that you avoid the use of wildcards to specify all available actions for a service. As a best practice, you should grant least privilege and narrowly scope permissions in a policy. For more information, see the section called “Grant minimum possible permissions”.

For the complete list of AWS IoT Greengrass actions, see Actions Defined by AWS IoT Greengrass in the *IAM User Guide*.

Resources

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Resource` JSON policy element specifies the object or objects to which the action applies. As a best practice, specify a resource using its Amazon Resource Name (ARN). For actions that don't support resource-level permissions, use a wildcard (*) to indicate that the statement applies to all resources.

```
"Resource": "*"
```

The following table contains the AWS IoT Greengrass resource ARNs that can be used in the `Resource` element of a policy statement. For a mapping of supported resource-level permissions for AWS IoT Greengrass actions, see Actions Defined by AWS IoT Greengrass in the *IAM User Guide*.

Some AWS IoT Greengrass actions (for example, some list operations), cannot be performed on a specific resource. In those cases, you must use the wildcard alone.

```
"Resource": "*"
```

To specify multiple resource ARNs in a statement, list them between brackets (`[ ]`) and separate them with commas, as follows:

```
"Resource": [
  "resource-arn1",
  "resource-arn2",
  "resource-arn3"
]
```

For more information about ARN formats, see Amazon Resource Names (ARNs) and AWS service namespaces in the *Amazon Web Services General Reference*.

Condition keys

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Condition` element specifies when statements execute based on defined criteria. You can create conditional expressions that use condition operators, such as equals or less than, to match the condition in the policy with values in the request. To see all AWS global condition keys, see AWS global condition context keys in the *IAM User Guide*.

Examples

To view examples of AWS IoT Greengrass identity-based policies, see the section called “Identity-based policy examples”.

Resource-based policies for AWS IoT Greengrass

AWS IoT Greengrass does not support resource-based policies.

Access control lists (ACLs)

AWS IoT Greengrass does not support ACLs.

Authorization based on AWS IoT Greengrass tags

You can attach tags to supported AWS IoT Greengrass resources or pass tags in a request to AWS IoT Greengrass. To control access based on tags, you provide tag information in the Condition element of a policy using the `aws:ResourceTag/${TagKey}`, `aws:RequestTag/${TagKey}`, or `aws:TagKeys` condition keys. For more information, see *Tag your resources*.

IAM roles for AWS IoT Greengrass

An IAM role is an entity within your AWS account that has specific permissions.

Using temporary credentials with AWS IoT Greengrass

Temporary credentials are used to sign in with federation, assume an IAM role, or to assume a cross-account role. You obtain temporary security credentials by calling AWS STS API operations such as AssumeRole or GetFederationToken.

On the Greengrass core, temporary credentials for the device role are made available to Greengrass components. If your components use the AWS SDK, you don't need to add logic to obtain the credentials because the AWS SDK does this for you.

Service-linked roles

AWS IoT Greengrass does not support service-linked roles.

Service roles

This feature allows a service to assume a service role on your behalf. This role allows the service to access resources in other services to complete an action on your behalf. Service roles appear in your IAM account and are owned by the account. This means that an IAM administrator can change the permissions for this role. However, doing so might break the functionality of the service.

AWS IoT Greengrass core devices use a service role to allow Greengrass components and Lambda functions to access some of your AWS resources on your behalf. For more information, see the section called “Authorize core devices to interact with AWS services”.

AWS IoT Greengrass uses a service role to access some of your AWS resources on your behalf. For more information, see Greengrass service role.

Identity-based policy examples for AWS IoT Greengrass

By default, IAM users and roles don't have permission to create or modify AWS IoT Greengrass resources. They also can't perform tasks using the AWS Management Console, AWS CLI, or AWS API. An IAM administrator must create IAM policies that grant users and roles permission to perform specific API operations on the specified resources they need. The administrator must then attach those policies to the IAM users or groups that require those permissions.

Policy best practices

Identity-based policies determine whether someone can create, access, or delete AWS IoT Greengrass resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started with AWS managed policies and move toward least-privilege permissions** – To get started granting permissions to your users and workloads, use the *AWS managed policies* that grant permissions for many common use cases. They are available in your AWS account. We recommend that you reduce permissions further by defining AWS customer managed policies that are specific to your use cases. For more information, see AWS managed policies or AWS managed policies for job functions in the *IAM User Guide*.
- **Apply least-privilege permissions** – When you set permissions with IAM policies, grant only the permissions required to perform a task. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as *least-privilege permissions*. For more information about using IAM to apply permissions, see Policies and permissions in IAM in the *IAM User Guide*.
- **Use conditions in IAM policies to further restrict access** – You can add a condition to your policies to limit access to actions and resources. For example, you can write a policy condition to specify that all requests must be sent using SSL. You can also use conditions to grant access to service actions if they are used through a specific AWS service, such as CloudFormation. For more information, see IAM JSON policy elements: Condition in the *IAM User Guide*.
- **Use IAM Access Analyzer to validate your IAM policies to ensure secure and functional permissions** – IAM Access Analyzer validates new and existing policies so that the policies adhere to the IAM policy language (JSON) and IAM best practices. IAM Access Analyzer provides more than 100 policy checks and actionable recommendations to help you author secure and functional policies. For more information, see Validate policies with IAM Access Analyzer in the *IAM User Guide*.

- **Require multi-factor authentication (MFA)** – If you have a scenario that requires IAM users or a root user in your AWS account, turn on MFA for additional security. To require MFA when API operations are called, add MFA conditions to your policies. For more information, see Secure API access with MFA in the *IAM User Guide*.

For more information about best practices in IAM, see Security best practices in IAM in the *IAM User Guide*.

Policy examples

The following example customer-defined policies grant permissions for common scenarios.

Examples

- Allow users to view their own permissions

To learn how to create an IAM identity-based policy using these example JSON policy documents, see Creating policies on the JSON tab in the *IAM User Guide*.

Allow users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
```

```
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

Authorize core devices to interact with AWS services

AWS IoT Greengrass core devices use the AWS IoT Core credentials provider to authorize calls to AWS services. The AWS IoT Core credentials provider enables devices to use their X.509 certificates as the unique device identity to authenticate AWS requests. This eliminates the need to store an AWS access key ID and secret access key on your AWS IoT Greengrass core devices. For more information, see Authorizing direct calls to AWS services in the *AWS IoT Core Developer Guide*.

When you run the AWS IoT Greengrass Core software, you can choose to provision the AWS resources that the core device requires. This includes the AWS Identity and Access Management (IAM) role that your core device assumes through the AWS IoT Core credentials provider. Use the `--provision true` argument to configure a role and policies that allow the core device to get temporary AWS credentials. This argument also configures an AWS IoT role alias that points to this IAM role. You can specify the name of the IAM role and AWS IoT role alias to use. If you specify `--provision true` without these other name parameters, the Greengrass core device creates and uses the following default resources:

- IAM role: `GreengrassV2TokenExchangeRole`

  This role has a policy named `GreengrassV2TokenExchangeRoleAccess` and a trust relationship that allows `credentials.iot.amazonaws.com` to assume the role. The policy includes the minimum permissions for the core device.

> **Important**
>
> This policy doesn't include access to files in S3 buckets. You must add permissions to the role to allow core devices to retrieve component artifacts from S3 buckets. For more information, see Allow access to S3 buckets for component artifacts.

- AWS IoT role alias: `GreengrassV2TokenExchangeRoleAlias`

  This role alias refers to the IAM role.

For more information, see Step 3: Install the AWS IoT Greengrass Core software.

You can also set the role alias for an existing core device. To do so, configure the `iotRoleAlias` configuration parameter of the Greengrass nucleus component.

You can acquire temporary AWS credentials for this IAM role to perform AWS operations in your custom components. For more information, see Interact with AWS services.

Topics

- Service role permissions for core devices
- Allow access to S3 buckets for component artifacts

Service role permissions for core devices

The role allows the following service to assume the role:

- `credentials.iot.amazonaws.com`

If you use the AWS IoT Greengrass Core software to create this role, it uses the following permissions policy to allow core devices to connect and send logs to AWS. The policy's name defaults to the name of the IAM role ending with `Access`. For example, if you use the default IAM role name, then this policy's name is `GreengrassV2TokenExchangeRoleAccess`.

Greengrass nucleus v2.5.0 and later

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:DescribeLogStreams",
        "s3:GetBucketLocation"
      ],
      "Resource": "*"
    }
  ]
}
```

v2.4.x

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:DescribeCertificate",
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:DescribeLogStreams",
        "s3:GetBucketLocation"
      ],
      "Resource": "*"
    }
  ]
```

```
}
```

Earlier than v2.4.0

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:DescribeCertificate",
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:DescribeLogStreams",
        "iot:Connect",
        "iot:Publish",
        "iot:Subscribe",
        "iot:Receive",
        "s3:GetBucketLocation"
      ],
      "Resource": "*"
    }
  ]
}
```

Allow access to S3 buckets for component artifacts

The default core device role doesn't allow core devices to access S3 buckets. To deploy components that have artifacts in S3 buckets, you must add the `s3:GetObject` permission to allow core devices to download component artifacts. You can add a new policy to the core device role to grant this permission.

To add a policy that allows access to component artifacts in Amazon S3

1. Create a file called `component-artifact-policy.json` and copy the following JSON into the file. This policy allows access to all files in an S3 bucket. Replace amzn-s3-demo-bucket with the name of the S3 bucket to allow the core device to access.

   JSON

   ```
   {
     "Version":"2012-10-17",
     "Statement": [
       {
         "Effect": "Allow",
         "Action": [
           "s3:GetObject"
         ],
         "Resource": "arn:aws:s3:::amzn-s3-demo-bucket/*"
       }
     ]
   }
   ```

2. Run the following command to create the policy from the policy document in `component-artifact-policy.json`.

   Linux or Unix

   ```
   aws iam create-policy \
     --policy-name MyGreengrassV2ComponentArtifactPolicy \
     --policy-document file://component-artifact-policy.json
   ```

   Windows Command Prompt (CMD)

   ```
   aws iam create-policy ^
     --policy-name MyGreengrassV2ComponentArtifactPolicy ^
     --policy-document file://component-artifact-policy.json
   ```

   PowerShell

   ```
   aws iam create-policy `
     --policy-name MyGreengrassV2ComponentArtifactPolicy `
   ```

```
  --policy-document file://component-artifact-policy.json
```

Copy the policy Amazon Resource Name (ARN) from the policy metadata in the output. You use this ARN to attach this policy to the core device role in the next step.

3. Run the following command to attach the policy to the core device role. Replace *GreengrassV2TokenExchangeRole* with the name of the role that you specified when you ran the AWS IoT Greengrass Core software. Then, replace the policy ARN with the ARN from the previous step.

   Linux or Unix

   ```
   aws iam attach-role-policy \
     --role-name GreengrassV2TokenExchangeRole \
     --policy-arn
    arn:aws:iam::123456789012:policy/MyGreengrassV2ComponentArtifactPolicy
   ```

   Windows Command Prompt (CMD)

   ```
   aws iam attach-role-policy ^
     --role-name GreengrassV2TokenExchangeRole ^
     --policy-arn
    arn:aws:iam::123456789012:policy/MyGreengrassV2ComponentArtifactPolicy
   ```

   PowerShell

   ```
   aws iam attach-role-policy `
     --role-name GreengrassV2TokenExchangeRole `
     --policy-arn
    arn:aws:iam::123456789012:policy/MyGreengrassV2ComponentArtifactPolicy
   ```

   If the command has no output, it succeeded, and your core device can access artifacts that you upload to this S3 bucket.

Minimal IAM policy for installer to provision resources

When you install the AWS IoT Greengrass Core software, you can provision required AWS resources, such as an AWS IoT thing and an IAM role for your device. You can also deploy local development

tools to the device. The installer requires AWS credentials so that it can perform these actions in your AWS account. For more information, see Install the AWS IoT Greengrass Core software.

The following example policy includes the minimum set of actions that the installer requires to provision these resources. These permissions are required if you specify the `--provision` argument for the installer. Replace *`account-id`* with your AWS account ID, and replace *`GreengrassV2TokenExchangeRole`* with the name of the token exchange role that you specify with the `--tes-role-name` installer argument.

> **Note**
>
> The `DeployDevTools` policy statement is required only if you specify the `--deploy-dev-tools` argument for the installer.

Greengrass nucleus v2.5.0 and later

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "CreateTokenExchangeRole",
            "Effect": "Allow",
            "Action": [
                "iam:AttachRolePolicy",
                "iam:CreatePolicy",
                "iam:CreateRole",
                "iam:GetPolicy",
                "iam:GetRole",
                "iam:PassRole"
            ],
            "Resource": [

 "arn:aws:iam::123456789012:role/GreengrassV2TokenExchangeRole",

 "arn:aws:iam::123456789012:policy/GreengrassV2TokenExchangeRoleAccess",
                "arn:aws:iam::aws:policy/GreengrassV2TokenExchangeRoleAccess"
            ]
        },
```

```
        {
            "Sid": "CreateIoTResources",
            "Effect": "Allow",
            "Action": [
                "iot:AddThingToThingGroup",
                "iot:AttachPolicy",
                "iot:AttachThingPrincipal",
                "iot:CreateKeysAndCertificate",
                "iot:CreatePolicy",
                "iot:CreateRoleAlias",
                "iot:CreateThing",
                "iot:CreateThingGroup",
                "iot:DescribeEndpoint",
                "iot:DescribeRoleAlias",
                "iot:DescribeThingGroup",
                "iot:GetPolicy"
            ],
            "Resource": "*"
        },
        {
            "Sid": "DeployDevTools",
            "Effect": "Allow",
            "Action": [
                "greengrass:CreateDeployment",
                "iot:CancelJob",
                "iot:CreateJob",
                "iot:DeleteThingShadow",
                "iot:DescribeJob",
                "iot:DescribeThing",
                "iot:DescribeThingGroup",
                "iot:GetThingShadow",
                "iot:UpdateJob",
                "iot:UpdateThingShadow"
            ],
            "Resource": "*"
        }
    ]
}
```

Earlier than v2.5.0

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "CreateTokenExchangeRole",
            "Effect": "Allow",
            "Action": [
                "iam:AttachRolePolicy",
                "iam:CreatePolicy",
                "iam:CreateRole",
                "iam:GetPolicy",
                "iam:GetRole",
                "iam:PassRole"
            ],
            "Resource": [
            "arn:aws:iam::123456789012:role/GreengrassV2TokenExchangeRole",
    "arn:aws:iam::123456789012:policy/GreengrassV2TokenExchangeRoleAccess",
                "arn:aws:iam::aws:policy/GreengrassV2TokenExchangeRoleAccess"
            ]
        },
        {
            "Sid": "CreateIoTResources",
            "Effect": "Allow",
            "Action": [
                "iot:AddThingToThingGroup",
                "iot:AttachPolicy",
                "iot:AttachThingPrincipal",
                "iot:CreateKeysAndCertificate",
                "iot:CreatePolicy",
                "iot:CreateRoleAlias",
                "iot:CreateThing",
                "iot:CreateThingGroup",
                "iot:DescribeEndpoint",
                "iot:DescribeRoleAlias",
                "iot:DescribeThingGroup",
                "iot:GetPolicy"
            ],
            "Resource": "*"
        },
```

```
        {
            "Sid": "DeployDevTools",
            "Effect": "Allow",
            "Action": [
                "greengrass:CreateDeployment",
                "iot:CancelJob",
                "iot:CreateJob",
                "iot:DeleteThingShadow",
                "iot:DescribeJob",
                "iot:DescribeThing",
                "iot:DescribeThingGroup",
                "iot:GetThingShadow",
                "iot:UpdateJob",
                "iot:UpdateThingShadow"
            ],
            "Resource": "*"
        }
    ]
}
```

Greengrass service role

The Greengrass service role is an AWS Identity and Access Management (IAM) service role that authorizes AWS IoT Greengrass to access resources from AWS services on your behalf. This role makes it possible for AWS IoT Greengrass to verify the identity of client devices and manage core device connectivity information.

> **Note**
>
> AWS IoT Greengrass V1 also uses this role to perform essential tasks. For more information, see Greengrass service role in the *AWS IoT Greengrass V1 Developer Guide*.

To allow AWS IoT Greengrass to access your resources, the Greengrass service role must be associated with your AWS account and specify AWS IoT Greengrass as a trusted entity. The role must include the AWSGreengrassResourceAccessRolePolicy managed policy or a custom policy that defines equivalent permissions for the AWS IoT Greengrass features that you use. AWS maintains this policy, which defines the set of permissions that AWS IoT Greengrass uses to access your AWS resources. For more information, see AWS managed policy: AWSGreengrassResourceAccessRolePolicy.

You can reuse the same Greengrass service role across AWS Regions, but you must associate it with your account in every AWS Region where you use AWS IoT Greengrass. If the service role isn't configured in the current AWS Region, core devices fail to verify client devices and fail to update connectivity information.

The following sections describe how to create and manage the Greengrass service role with the AWS Management Console or AWS CLI.

Topics

- Manage the Greengrass service role (console)
- Manage the Greengrass service role (CLI)
- See also

> **Note**
>
> In addition to the service role that authorizes service-level access, you assign a *token exchange role* to Greengrass core devices. The token exchange role is a separate IAM role that controls how Greengrass components and Lambda functions on the core device can access AWS services. For more information, see Authorize core devices to interact with AWS services.

Manage the Greengrass service role (console)

The AWS IoT console makes it easy to manage your Greengrass service role. For example, when you configure client device discovery for a core device, the console checks whether your AWS account is attached to a Greengrass service role in the current AWS Region. If not, the console can create and configure a service role for you. For more information, see the section called “Create the Greengrass service role”.

You can use the console for the following role management tasks:

Topics

- Find your Greengrass service role (console)
- Create the Greengrass service role (console)
- Change the Greengrass service role (console)

- Detach the Greengrass service role (console)

> **Note**
>
> The user who is signed in to the console must have permissions to view, create, or change the service role.

Find your Greengrass service role (console)

Use the following steps to find the service role that AWS IoT Greengrass uses in the current AWS Region.

1. Navigate to the AWS IoT console.
2. In the navigation pane, choose **Settings**.
3. Scroll to the **Greengrass service role** section to see your service role and its policies.

   If you don't see a service role, the console can create or configure one for you. For more information, see Create the Greengrass service role.

Create the Greengrass service role (console)

The console can create and configure a default Greengrass service role for you. This role has the following properties.

Property	Value
Name	`Greengrass_ServiceRole`
Trusted entity	`AWS service: greengrass`
Policy	AWSGreengrassResourceAccessRolePolicy

> **Note**
>
> If you create this role with the AWS IoT Greengrass V1 device setup script, the role name is `GreengrassServiceRole_`*`random-string`*.

When you configure client device discovery for a core device, the console checks whether a Greengrass service role is associated with your AWS account in the current AWS Region. If not, the console prompts you to allow AWS IoT Greengrass to read and write to AWS services on your behalf.

If you grant permission, the console checks whether a role named `Greengrass_ServiceRole` exists in your AWS account.

- If the role exists, the console attaches the service role to your AWS account in the current AWS Region.
- If the role doesn't exist, the console creates a default Greengrass service role and attaches it to your AWS account in the current AWS Region.

> **Note**
>
> If you want to create a service role with custom role policies, use the IAM console to create or modify the role. For more information, see Creating a role to delegate permissions to an AWS service or Modifying a role in the *IAM User Guide*. Make sure that the role grants permissions that are equivalent to the `AWSGreengrassResourceAccessRolePolicy` managed policy for the features and resources that you use. We recommend that you also include the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in your trust policy to help prevent the *confused deputy* security problem. The condition context keys restrict access to allow only those requests that come from the specified account and Greengrass workspace. For more information about the confused deputy problem, see Cross-service confused deputy prevention.
> If you create a service role, return to the AWS IoT console and attach the role to your AWS account. You can do this under **Greengrass service role** on the **Settings** page.

Change the Greengrass service role (console)

Use the following procedure to choose a different Greengrass service role to attach to your AWS account in the AWS Region currently selected in the console.

1. Navigate to the AWS IoT console.
2. In the navigation pane, choose **Settings**.
3. Under **Greengrass service role**, choose **Change role**.

The **Update Greengrass service role** dialog box opens and shows the IAM roles in your AWS account that define AWS IoT Greengrass as a trusted entity.

4. Choose the Greengrass service role to attach.
5. Choose **Attach role**.

Detach the Greengrass service role (console)

Use the following procedure to detach the Greengrass service role from your AWS account in the current AWS Region. This revokes permissions for AWS IoT Greengrass to access AWS services in the current AWS Region.

> **Important**
>
> Detaching the service role might interrupt active operations.

1. Navigate to the AWS IoT console.
2. In the navigation pane, choose **Settings**.
3. Under **Greengrass service role**, choose **Detach role**.
4. In the confirmation dialog box, choose **Detach**.

> **Note**
>
> If you no longer need the role, you can delete it in the IAM console. For more information, see Deleting roles or instance profiles in the *IAM User Guide*.
> Other roles might allow AWS IoT Greengrass to access your resources. To find all roles that allow AWS IoT Greengrass to assume permissions on your behalf, in the IAM console, on the **Roles** page, look for roles that include **AWS service: greengrass** in the **Trusted entities** column.

Manage the Greengrass service role (CLI)

In the following procedures, we assume that the AWS Command Line Interface is installed and configured to use your AWS account. For more information, see Installing, updating, and

uninstalling the AWS CLI and Configuring the AWS CLI in the *AWS Command Line Interface User Guide*.

You can use the AWS CLI for the following role management tasks:

Topics

- Get the Greengrass service role (CLI)
- Create the Greengrass service role (CLI)
- Remove the Greengrass service role (CLI)

Get the Greengrass service role (CLI)

Use the following procedure to find out if a Greengrass service role is associated with your AWS account in an AWS Region.

- Get the service role. Replace *region* with your AWS Region (for example, `us-west-2`).

  ```
  aws greengrassv2 get-service-role-for-account --region region
  ```

  If a Greengrass service role is already associated with your account, the request returns the following role metadata.

  ```
  {
    "associatedAt": "timestamp",
    "roleArn": "arn:aws:iam::account-id:role/path/role-name"
  }
  ```

  If the request doesn't return role metadata, then you must create the service role (if it doesn't exist) and associate it with your account in the AWS Region.

Create the Greengrass service role (CLI)

Use the following steps to create a role and associate it with your AWS account.

To create the service role using IAM

1. Create a role with a trust policy that allows AWS IoT Greengrass to assume the role. This example creates a role named `Greengrass_ServiceRole`, but you can use a different

name. We recommend that you also include the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in your trust policy to help prevent the *confused deputy* security problem. The condition context keys restrict access to allow only those requests that come from the specified account and Greengrass workspace. For more information about the confused deputy problem, see Cross-service confused deputy prevention.

Linux or Unix

```
aws iam create-role --role-name Greengrass_ServiceRole --assume-role-policy-
document '{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "greengrass.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "ArnLike": {
          "aws:SourceArn": "arn:aws:greengrass:region:account-id:*"
        },
        "StringEquals": {
          "aws:SourceAccount": "account-id"
        }
      }
    }
  ]
}'
```

Windows Command Prompt (CMD)

```
aws iam create-role --role-name Greengrass_ServiceRole --assume-role-
policy-document "{\"Version\":\"2012-10-17\",       \"Statement\":[{\"Effect
\":\"Allow\",\"Principal\":{\"Service\":\"greengrass.amazonaws.com\"},
\"Action\":\"sts:AssumeRole\",\"Condition\":{\"ArnLike\":{\"aws:SourceArn
\":\"arn:aws:greengrass:region:account-id:*\"},\"StringEquals\":
{\"aws:SourceAccount\":\"account-id\"}}}]}"
```

PowerShell

```
aws iam create-role --role-name Greengrass_ServiceRole --assume-role-policy-
document '{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "greengrass.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "ArnLike": {
          "aws:SourceArn": "arn:aws:greengrass:region:account-id:*"
        },
        "StringEquals": {
          "aws:SourceAccount": "account-id"
        }
      }
    }
  ]
}'
```

2. Copy the role ARN from the role metadata in the output. You use the ARN to associate the role with your account.
3. Attach the `AWSGreengrassResourceAccessRolePolicy` policy to the role.

```
aws iam attach-role-policy --role-name Greengrass_ServiceRole --policy-arn
 arn:aws:iam::aws:policy/service-role/AWSGreengrassResourceAccessRolePolicy
```

To associate the service role with your AWS account

- Associate the role with your account. Replace *`role-arn`* with the service role ARN and *`region`* with your AWS Region (for example, `us-west-2`).

```
aws greengrassv2 associate-service-role-to-account --role-arn role-arn --
region region
```

If successful, the request returns the following response.

```
{
  "associatedAt": "timestamp"
}
```

Remove the Greengrass service role (CLI)

Use the following steps to disassociate the Greengrass service role from your AWS account.

- Disassociate the service role from your account. Replace *region* with your AWS Region (for example, `us-west-2`).

  ```
  aws greengrassv2 disassociate-service-role-from-account --region region
  ```

  If successful, the following response is returned.

  ```
  {
    "disassociatedAt": "timestamp"
  }
  ```

  > **Note**
  >
  > You should delete the service role if you're not using it in any AWS Region. First use **delete-role-policy** to detach the `AWSGreengrassResourceAccessRolePolicy` managed policy from the role, and then use **delete-role** to delete the role. For more information, see Deleting roles or instance profiles in the *IAM User Guide*.

See also

- Creating a role to delegate permissions to an AWS service in the *IAM User Guide*
- Modifying a role in the *IAM User Guide*
- Deleting roles or instance profiles in the *IAM User Guide*
- AWS IoT Greengrass commands in the *AWS CLI Command Reference*
  - associate-service-role-to-account
  - disassociate-service-role-from-account

- get-service-role-for-account
- IAM commands in the *AWS CLI Command Reference*
  - attach-role-policy
  - create-role
  - delete-role
  - delete-role-policy

AWS managed policies for AWS IoT Greengrass

An AWS managed policy is a standalone policy that is created and administered by AWS. AWS managed policies are designed to provide permissions for many common use cases so that you can start assigning permissions to users, groups, and roles.

Keep in mind that AWS managed policies might not grant least-privilege permissions for your specific use cases because they're available for all AWS customers to use. We recommend that you reduce permissions further by defining customer managed policies that are specific to your use cases.

You cannot change the permissions defined in AWS managed policies. If AWS updates the permissions defined in an AWS managed policy, the update affects all principal identities (users, groups, and roles) that the policy is attached to. AWS is most likely to update an AWS managed policy when a new AWS service is launched or new API operations become available for existing services.

For more information, see AWS managed policies in the *IAM User Guide*.

Topics

- AWS managed policy: AWSGreengrassFullAccess
- AWS managed policy: AWSGreengrassReadOnlyAccess
- AWS managed policy: AWSGreengrassResourceAccessRolePolicy
- AWS IoT Greengrass updates to AWS managed policies

AWS managed policy: AWSGreengrassFullAccess

You can attach the `AWSGreengrassFullAccess` policy to your IAM identities.

This policy grants administrative permissions that allow a principal full access to all AWS IoT Greengrass actions.

Permissions details

This policy includes the following permissions:

- `greengrass` – Allows principals full access to all AWS IoT Greengrass actions.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "greengrass:*"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS managed policy: AWSGreengrassReadOnlyAccess

You can attach the `AWSGreengrassReadOnlyAccess` policy to your IAM identities.

This policy grants read-only permissions that allow a principal to view, but not modify, information in AWS IoT Greengrass. For example, principals with these permissions can view the list of components deployed to a Greengrass core device, but can't create a deployment to change the components that run on that device.

Permissions details

This policy includes the following permissions:

- `greengrass` – Allows principals to perform actions that return either a list of items or details about an item. This includes API operations that start with `List` or `Get`.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "greengrass:List*",
                "greengrass:Get*"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS managed policy: AWSGreengrassResourceAccessRolePolicy

You can attach the `AWSGreengrassResourceAccessRolePolicy` policy to your IAM entities. AWS IoT Greengrass also attaches this policy to a service role that allows AWS IoT Greengrass to perform actions on your behalf. For more information, see Greengrass service role.

This policy grants administrative permissions that allow AWS IoT Greengrass to perform essential tasks, such as retrieving your Lambda functions, managing AWS IoT device shadows, and verifying Greengrass client devices.

Permissions details

This policy includes the following permissions.

- `greengrass` – Manage Greengrass resources.
- `iot` (`*Shadow`) – Manage AWS IoT shadows that have the following special identifiers in their names. These permissions are required so that AWS IoT Greengrass can communicate with core devices.
  - `*-gci` – AWS IoT Greengrass uses this shadow to store core device connectivity information, so client devices can discover and connect to core devices.
  - `*-gcm` – AWS IoT Greengrass V1 uses this shadow to notify the core device that the Greengrass group's certificate authority (CA) certificate has rotated.

  - `*-gda` – AWS IoT Greengrass V1 uses this shadow to notify the core device of a deployment.
  - `GG_*` – Unused.
- `iot` (`DescribeThing` and `DescribeCertificate`) – Retrieve information about AWS IoT things and certificates. These permissions are required so that AWS IoT Greengrass can verify client devices that connect to a core device. For more information, see Interact with local IoT devices.
- `lambda` – Retrieve information about AWS Lambda functions. This permission is required so that AWS IoT Greengrass V1 can deploy Lambda functions to Greengrass cores. For more information, see Run Lambda function on the AWS IoT Greengrass core in the *AWS IoT Greengrass V1 Developer Guide*.
- `secretsmanager` – Retrieve the value of AWS Secrets Manager secrets whose names start with `greengrass-`. This permission is required so that AWS IoT Greengrass V1 can deploy Secrets Manager secrets to Greengrass cores. For more information, see Deploy secrets to the AWS IoT Greengrass core in the *AWS IoT Greengrass V1 Developer Guide*.
- `s3` – Retrieve files objects from S3 buckets whose names contain `greengrass` or `sagemaker`. These permissions are required so that AWS IoT Greengrass V1 can deploy machine learning resources that you store in S3 buckets. For more information, see Machine learning resources in the *AWS IoT Greengrass V1 Developer Guide*.
- `sagemaker` – Retrieve information about Amazon SageMaker AI machine learning inference models. This permission is required so that AWS IoT Greengrass V1 can deploy ML models to Greengrass cores. For more information, see Perform machine learning inference in the *AWS IoT Greengrass V1 Developer Guide*.

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "AllowGreengrassAccessToShadows",
            "Action": [
                "iot:DeleteThingShadow",
                "iot:GetThingShadow",
                "iot:UpdateThingShadow"
            ],
            "Effect": "Allow",
```

```
            "Resource": [
                "arn:aws:iot:*:*:thing/GG_*",
                "arn:aws:iot:*:*:thing/*-gcm",
                "arn:aws:iot:*:*:thing/*-gda",
                "arn:aws:iot:*:*:thing/*-gci"
            ]
        },
        {
            "Sid": "AllowGreengrassToDescribeThings",
            "Action": [
                "iot:DescribeThing"
            ],
            "Effect": "Allow",
            "Resource": "arn:aws:iot:*:*:thing/*"
        },
        {
            "Sid": "AllowGreengrassToDescribeCertificates",
            "Action": [
                "iot:DescribeCertificate"
            ],
            "Effect": "Allow",
            "Resource": "arn:aws:iot:*:*:cert/*"
        },
        {
            "Sid": "AllowGreengrassToCallGreengrassServices",
            "Action": [
                "greengrass:*"
            ],
            "Effect": "Allow",
            "Resource": "*"
        },
        {
            "Sid": "AllowGreengrassToGetLambdaFunctions",
            "Action": [
                "lambda:GetFunction",
                "lambda:GetFunctionConfiguration"
            ],
            "Effect": "Allow",
            "Resource": "*"
        },
        {
            "Sid": "AllowGreengrassToGetGreengrassSecrets",
            "Action": [
                "secretsmanager:GetSecretValue"
```

```
            ],
            "Effect": "Allow",
            "Resource": "arn:aws:secretsmanager:*:*:secret:greengrass-*"
        },
        {
            "Sid": "AllowGreengrassAccessToS3Objects",
            "Action": [
                "s3:GetObject"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:s3:::*Greengrass*",
                "arn:aws:s3:::*GreenGrass*",
                "arn:aws:s3:::*greengrass*",
                "arn:aws:s3:::*Sagemaker*",
                "arn:aws:s3:::*SageMaker*",
                "arn:aws:s3:::*sagemaker*"
            ]
        },
        {
            "Sid": "AllowGreengrassAccessToS3BucketLocation",
            "Action": [
                "s3:GetBucketLocation"
            ],
            "Effect": "Allow",
            "Resource": "*"
        },
        {
            "Sid": "AllowGreengrassAccessToSageMakerTrainingJobs",
            "Action": [
                "sagemaker:DescribeTrainingJob"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:sagemaker:*:*:training-job/*"
            ]
        }
    ]
}
```

AWS IoT Greengrass updates to AWS managed policies

You can view details about updates to AWS managed policies for AWS IoT Greengrass from the time this service began tracking these changes. For automatic alerts about changes to this page, subscribe to the RSS feed on the AWS IoT Greengrass V2 document history page.

Change	Description	Date
AWS IoT Greengrass started tracking changes	AWS IoT Greengrass started tracking changes for its AWS managed policies.	July 2, 2021

Cross-service confused deputy prevention

The confused deputy problem is a security issue where an entity that doesn't have permission to perform an action can coerce a more-privileged entity to perform the action. In AWS, cross-service impersonation can result in the confused deputy problem. Cross-service impersonation can occur when one service (the *calling service*) calls another service (the *called service*). The calling service can be manipulated to use its permissions to act on another customer's resources in a way it should not otherwise have permission to access. To prevent this, AWS provides tools that help you protect your data for all services with service principals that have been given access to resources in your account.

We recommend using the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in resource policies to limit the permissions that AWS IoT Greengrass gives another service to the resource. If you use both global condition context keys, the `aws:SourceAccount` value and the account in the `aws:SourceArn` value must use the same account ID when used in the same policy statement.

The value of `aws:SourceArn` must be the Greengrass customer resource that is associated with the `sts:AssumeRole` request.

The most effective way to protect against the confused deputy problem is to use the `aws:SourceArn` global condition context key with the full ARN of the resource. If you don't know the full ARN of the resource or if you are specifying multiple resources, use the `aws:SourceArn` global context condition key with wildcards (*) for the unknown portions of the ARN. For example, `arn:aws:greengrass::`*`account-id`*`:*`.

For an example of a policy that uses the `aws:SourceArn` and `aws:SourceAccount` global condition context keys, see Create the Greengrass service role.

Troubleshooting identity and access issues for AWS IoT Greengrass

Use the following information to help you diagnose and fix common issues that you might encounter when working with AWS IoT Greengrass and IAM.

Issues

- I'm not authorized to perform an action in AWS IoT Greengrass
- I'm not authorized to perform iam:PassRole
- I'm an administrator and want to allow others to access AWS IoT Greengrass
- I want to allow people outside of my AWS account to access my AWS IoT Greengrass resources

For general troubleshooting help, see *Troubleshooting*.

I'm not authorized to perform an action in AWS IoT Greengrass

If you receive an error that states you're not authorized to perform an action, you must contact your administrator for assistance. Your administrator is the person who provided you with your user name and password.

The following example error occurs when the `mateojackson` IAM user tries to view details about a core device, but does not have `greengrass:GetCoreDevice` permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to
 perform: greengrass:GetCoreDevice on resource: arn:aws:greengrass:us-
west-2:123456789012:coreDevices/MyGreengrassCore
```

In this case, Mateo asks his administrator to update his policies to allow him to access the `arn:aws:greengrass:us-west-2:123456789012:coreDevices/MyGreengrassCore` resource using the `greengrass:GetCoreDevice` action.

The following are general IAM issues that you might encounter when working with AWS IoT Greengrass.

I'm not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the `iam:PassRole` action, your policies must be updated to allow you to pass a role to AWS IoT Greengrass.

Some AWS services allow you to pass an existing role to that service instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named `marymajor` tries to use the console to perform an action in AWS IoT Greengrass. However, the action requires the service to have permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

In this case, Mary's policies must be updated to allow her to perform the `iam:PassRole` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I'm an administrator and want to allow others to access AWS IoT Greengrass

To allow others to access AWS IoT Greengrass, you must grant permission to the people or applications that need access. If you are using AWS IAM Identity Center to manage people and applications, you assign permission sets to users or groups to define their level of access. Permission sets automatically create and assign IAM policies to IAM roles that are associated with the person or application. For more information, see Permission sets in the *AWS IAM Identity Center User Guide*.

If you are not using IAM Identity Center, you must create IAM entities (users or roles) for the people or applications that need access. You must then attach a policy to the entity that grants them the correct permissions in AWS IoT Greengrass. After the permissions are granted, provide the credentials to the user or application developer. They will use those credentials to access AWS. To learn more about creating IAM users, groups, policies, and permissions, see IAM Identities and Policies and permissions in IAM in the *IAM User Guide*.

I want to allow people outside of my AWS account to access my AWS IoT Greengrass resources

You can create an IAM role that users in other accounts or people outside of your organization can use to access your AWS resources. You can specify the who is trusted to assume the role. For more information, see Providing access to an IAM user in another AWS account that you own and Providing access to AWS accounts owned by third parties in the *IAM User Guide*.

AWS IoT Greengrass doesn't support cross-account access based on resource-based policies or access control lists (ACLs).

Allow device traffic through a proxy or firewall

Greengrass core devices and Greengrass components perform outbound requests to AWS services and other websites. As a security measure, you might limit outbound traffic to a small range of endpoints and ports. You can use the following information about endpoints and ports to limit device traffic through a proxy, firewall, or Amazon VPC security group. For more information about how to configure a core device to use a proxy, see Connect on port 443 or through a network proxy.

Topics

- Endpoints for basic operation
- Endpoints for installation with automatic provisioning
- Endpoints for AWS-provided components

Endpoints for basic operation

Greengrass core devices use the following endpoints and ports for basic operation.

Retrieve AWS IoT endpoints

Get the AWS IoT endpoints for your AWS account, and save them to use later. Your device uses these endpoints to connect to AWS IoT. Do the following:

1. Get the AWS IoT data endpoint for your AWS account.

   ```
   aws iot describe-endpoint --endpoint-type iot:Data-ATS
   ```

   The response looks similar to the following example, if the request succeeds.

```
{
  "endpointAddress": "device-data-prefix-ats.iot.us-west-2.amazonaws.com"
}
```

2. Get the AWS IoT credentials endpoint for your AWS account.

```
aws iot describe-endpoint --endpoint-type iot:CredentialProvider
```

The response looks similar to the following example, if the request succeeds.

```
{
  "endpointAddress": "device-credentials-prefix.credentials.iot.us-
west-2.amazonaws.com"
}
```

Endpoint	Port	Required	Description
greengrass-ats.iot. *region*.amazonaws.com	8443 or 443	Yes	Used for data plane operations, such as installing deployments and working with client devices.
device-data-prefix -ats.iot. *region*.amazonaws.com	MQTT: 8883 or 443 HTTPS: 8443 or 443	Yes	Used for data plane operations for device management, such as MQTT

Endpoint	Port	Required	Description
			communication and shadow sync with AWS IoT Core.
device-credentials-prefix .credentials.iot. *region*.amazonaws.com	443	Yes	Used to acquire AWS credentials, which the core device uses to download component artifacts from Amazon S3 and perform other operations. For more information, see Authorize core devices to interact with AWS services.

Endpoint	Port	Required	Description

`*.s3.amazonaws.com`
 	
`*.s3.`*`region`*`.amazonaws.com`
	443	Yes	Used for deployments. This format includes the * character, because endpoint prefixes are controlled internally and might change at any time.

Endpoint	Port	Required	Description
`data.iot.` *`region`*`.amazonaws.com`	443	No	Required if the core device runs a version of the Greengrass nucleus earlier than v2.4.0 and is configured to use a network proxy. The core device uses this endpoint for MQTT communication with AWS IoT Core when behind a proxy. For more information, see Configure a network proxy.

Endpoints for installation with automatic provisioning

Greengrass core devices use the following endpoints and ports when you install the AWS IoT Greengrass Core software with automatic resource provisioning.

Endpoint	Port	Required	Description
`iot.region.amazonaws.com`	443	Yes	Used to create AWS IoT resources and retrieve information about existing AWS IoT resources.
`iam.amazonaws.com`	443	Yes	Used to create IAM resources and retrieve information about existing IAM resources.
`sts.region.amazonaws.com`	443	Yes	Used to get the ID of your AWS account.

Endpoint	Port	Required	Description
`greengrass. region.amazonaws.com`	443	No	Required if you use the `--deploy-dev-tools` argument to deploy the Greengrass CLI component to the core device.

Endpoints for AWS-provided components

Greengrass core devices use additional endpoints depending on which software components they run. You can find the endpoints that each AWS-provided component requires in the **Requirements** section on each component's page in this developer guide. For more information, see AWS-provided components.

Compliance validation for AWS IoT Greengrass

Our new AWS sign-up experience is not designed for regulated workloads. If you're using our new AWS sign-up experience, but you want to use AWS for regulated workloads, you can sign up for AWS (advanced) or activate advanced features for your AWS environment.

To learn whether an AWS service is within the scope of specific compliance programs, see AWS services in Scope by Compliance Program and choose the compliance program that you are interested in. For general information, see AWS Compliance Programs.

You can download third-party audit reports using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

Your compliance responsibility when using AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. For more information about your compliance responsibility when using AWS services, see AWS Security Documentation.

FIPS endpoints

AWS IoT Greengrass supports the use of FIPS (Federal Information Processing Standard (FIPS) 140-2) endpoints. When FIPS mode is enabled, all data transmissions, including both HTTP and MQTT protocols, to AWS Cloud services should invoke and establish connections with the corresponding FIPS-compliant endpoints (FIPS - Amazon Web Services (AWS)).

MQTT communications to AWS IoT utilize the IoT dataplane FIPS endpoint (Connecting to AWS IoT FIPS endpoints - AWS IoT Core) and the AWS-developed FIPS-compliant cryptographic library aws-lc.

For HTTP communications in Greengrass:

- For nucleus and plugin components, all SDK HTTP clients are configured with FIPS endpoints by setting the system property AWS_USE_FIPS_ENDPOINT to true;
- For generic components, all components start with the system property AWS_USE_FIPS_ENDPOINT set to true. This process ensures that the SDK HTTP clients used by these generic components send requests to FIPS-compliant endpoints.

> **Note**
>
> In the case of Stream manager, Nucleus passes the environment variable AWS_GG_FIPS_MODE. This environment variable allows the HTTP clients utilized within the Stream Manager to identify and connect to the corresponding FIPS-compliant endpoint.

AWS IoT Greengrass offers two methods to enable FIPS mode: provisioning and deployment. To activate the FIPS mode, you have to set the configuration parameter `fipsMode` to true, Nucleus then sets the system property AWS_USE_FIPS_ENDPOINT to true and propagate it as an environment variable to all other components. Additionally, AWS IoT Greengrass will download a root CA certificate (CA3) and append it to the existing rootCA.pem (or AmazonRootCA1.pem)

file. If you enable FIPS through a new deployment, Nucleus will restart to ensure that the system property takes effect after enabling FIPS mode.

Apart from configuring the `fipsMode` parameter, you must also configure the `iotDataEndpoint`, `iotCredEndpoint` and `greengrassDataEndpoint` parameters. For more information, see the relevant document below.

Enable FIPS endpoints with deployment

Get the AWS IoT endpoints for your AWS account, and save them to use later. Your device uses these endpoints to connect to AWS IoT. There are two endpoints required, the `iotDataEndpoint` and the `iotCredEndpoint`. Do the following:

1. Get the FIPS data endpoint for your region in the AWS IoT Core FIPS data plane endpoints. The FIPS data endpoint for your AWS account should look like this: *`data.iot-fips.us-west-2.amazonaws.com`*
2. Get the FIPS credentials endpoint for your region in the AWS IoT Core FIPS data plane endpoints. The FIPS credentials endpoint for your AWS account should look like this: *`data.credentials.iot-fips.us-west-2.amazonaws.com`*

Then, to enable FIPS with a deployment, you need to apply the following configuration to Nucleus. The configuration to merge on the deployment is as follows.

Console

Configuration to merge

```
{
  "fipsMode": "true",
  "iotDataEndpoint": "data.iot-fips.us-west-2.amazonaws.com",
  "greengrassDataPlaneEndpoint": "iotData",
  "iotCredEndpoint": "data.credentials.iot-fips.us-west-2.amazonaws.com"
}
```

AWS CLI

The following command creates a deployment to a core device.

```
aws greengrassv2 create-deployment --cli-input-json file://dashboard-deployment.json
```

The `dashboard-deployment.json` file contains the following JSON document.

```
{
  "targetArn": "arn:aws:iot:us-west-2:123456789012:thing/MyGreengrassCore",
  "deploymentName": "Deployment for MyGreengrassCore",
  "components": {
    "aws.greengrass.Nucleus": {
      "componentVersion": "2.13.0",
      "configurationUpdate": {
        "merge":{\"fipsMode\":\"true\",\"iotDataEndpoint\":\"data.iot-fips.us-
west-2.amazonaws.com\",\"greengrassDataPlaneEndpoint\":\"iotData\",\"iotCredEndpoint
\":\"data.credentials.iot-fips.us-west-2.amazonaws.com\"}"
      }
    }
  }
}
```

Greengrass CLI

The following Greengrass CLI command creates a local deployment on a core device.

```
sudo greengrass-cli deployment create \
  --recipeDir recipes \
  --artifactDir artifacts \
  --merge "aws.greengrass.Nucleus=2.13.0" \
  --update-config dashboard-configuration.json
```

The `dashboard-configuration.json` file contains the following JSON document.

```
{
  "aws.greengrass.Nucleus": {
    "MERGE": {
       "fipsMode": "true",
       "iotDataEndpoint": "data.iot-fips.us-west-2.amazonaws.com",
       "greengrassDataPlaneEndpoint": "iotData",
       "iotCredEndpoint": "data.credentials.iot-fips.us-west-2.amazonaws.com"

    }
  }
```

```
}
```

Install Nucleus with FIPS endpoints with manual resource provisioning

Manually provision AWS resources for AWS IoT Greengrass V2 core devices with FIPS endpoints

> **Important**
>
> Before you download the AWS IoT Greengrass Core software, check that your core device meets the requirements to install and run the AWS IoT Greengrass Core software v2.0.

Topics

- Retrieve AWS IoT endpoints
- Create an AWS IoT thing
- Create the thing certificate
- Create a token exchange role
- Download certificates to the device
- Set up the device environment
- Download the AWS IoT Greengrass Core software
- Install the AWS IoT Greengrass Core software

Retrieve AWS IoT endpoints

Get the AWS IoT endpoints for your AWS account, and save them to use later. Your device uses these endpoints to connect to AWS IoT. There are two endpoints required, the `iotDataEndpoint` and the `iotCredEndpoint`. Do the following:

1. Get the FIPS data endpoint for your region in the AWS IoT Core FIPS data plane endpoints. The FIPS data endpoint for your AWS account should look like this: *`data.iot-fips.us-west-2.amazonaws.com`*
2. Get the FIPS credentials endpoint for your region in the AWS IoT Core FIPS data plane endpoints. The FIPS credentials endpoint for your AWS account should look like this: *`data.credentials.iot-fips.us-west-2.amazonaws.com`*

Create an AWS IoT thing

AWS IoT *things* represent devices and logical entities that connect to AWS IoT. Greengrass core devices are AWS IoT things. When you register a device as an AWS IoT thing, that device can use a digital certificate to authenticate with AWS.

In this section, you create an AWS IoT thing that represents your device.

To create an AWS IoT thing

1. Create an AWS IoT thing for your device. On your development computer, run the following command.

   - Replace *MyGreengrassCore* with the thing name to use. This name is also the name of your Greengrass core device.

     > **Note**
     >
     > The thing name can't contain colon (:) characters.

   ```
   aws iot create-thing --thing-name MyGreengrassCore
   ```

   The response looks similar to the following example, if the request succeeds.

   ```
   {
     "thingName": "MyGreengrassCore",
     "thingArn": "arn:aws:iot:us-west-2:123456789012:thing/MyGreengrassCore",
     "thingId": "8cb4b6cd-268e-495d-b5b9-1713d71dbf42"
   }
   ```

2. (Optional) Add the AWS IoT thing to a new or existing thing group. You use thing groups to manage fleets of Greengrass core devices. When you deploy software components to your devices, you can target individual devices or groups of devices. You can add a device to a thing group with an active Greengrass deployment to deploy that thing group's software components to the device. Do the following:

   a. (Optional) Create an AWS IoT thing group.

      - Replace *MyGreengrassCoreGroup* with the name of the thing group to create.

> **Note**
>
> The thing group name can't contain colon (:) characters.

```
aws iot create-thing-group --thing-group-name MyGreengrassCoreGroup
```

The response looks similar to the following example, if the request succeeds.

```
{
  "thingGroupName": "MyGreengrassCoreGroup",
  "thingGroupArn": "arn:aws:iot:us-west-2:123456789012:thinggroup/
MyGreengrassCoreGroup",
  "thingGroupId": "4df721e1-ff9f-4f97-92dd-02db4e3f03aa"
}
```

b. Add the AWS IoT thing to a thing group.

- Replace *MyGreengrassCore* with the name of your AWS IoT thing.
- Replace *MyGreengrassCoreGroup* with the name of the thing group.

```
aws iot add-thing-to-thing-group --thing-name MyGreengrassCore --thing-group-
name MyGreengrassCoreGroup
```

The command doesn't have any output if the request succeeds.

Create the thing certificate

When you register a device as an AWS IoT thing, that device can use a digital certificate to authenticate with AWS. This certificate allows the device to communicate with AWS IoT and AWS IoT Greengrass.

In this section, you create and download certificates that your device can use to connect to AWS.

If you want to configure the AWS IoT Greengrass Core software to use a hardware security module (HSM) to securely store the private key and certificate, follow the steps to create the certificate from a private key in an HSM. Otherwise, follow the steps to create the certificate and private key

in the AWS IoT service. The hardware security feature is available on Linux devices only. For more information about hardware security and requirements to use it, see Hardware security integration.

Create the certificate and private key in the AWS IoT service

To create the thing certificate

1. Create a folder where you download the certificates for the AWS IoT thing.

```
mkdir greengrass-v2-certs
```

2. Create and download the certificates for the AWS IoT thing.

```
aws iot create-keys-and-certificate --set-as-active --certificate-pem-outfile
 greengrass-v2-certs/device.pem.crt --public-key-outfile greengrass-v2-certs/
public.pem.key --private-key-outfile greengrass-v2-certs/private.pem.key
```

   The response looks similar to the following example, if the request succeeds.

```
{
  "certificateArn": "arn:aws:iot:us-west-2:123456789012:cert/
aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4",
  "certificateId":
 "aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4",
  "certificatePem": "-----BEGIN CERTIFICATE-----
MIICiTCCAfICCQD6m7oRw0uXOjANBgkqhkiG9w
 0BAQUFADCBiDELMAkGA1UEBhMCVVMxCzAJBgNVBAgTAldBMRAwDgYDVQQHEwdTZ
 WF0dGxlMQ8wDQYDVQQKEwZBbWF6b24xFDASBgNVBAsTC0lBTSBDb25zb2xlMRIw
 EAYDVQQDEwlUZXN0Q2lsYWMxHzAdBgkqhkiG9w0BCQEWEG5vb25lQGFtYXpvbi5
 jb20wHhcNMTEwNDI1MjA0NTIxWhcNMTIwNDI0MjA0NTIxWjCBiDELMAkGA1UEBh
 MCVVMxCzAJBgNVBAgTAldBMRAwDgYDVQQHEwdTZWF0dGxlMQ8wDQYDVQQKEwZBb
 WF6b24xFDASBgNVBAsTC0lBTSBDb25zb2xlMRIwEAYDVQQDEwlUZXN0Q2lsYWMx
 HzAdBgkqhkiG9w0BCQEWEG5vb25lQGFtYXpvbi5jb20wgZ8wDQYJKoZIhvcNAQE
 BBQADgY0AMIGJAoGBAMaK0dn+a4GmWIWJ21uUSfwfEvySWtC2XADZ4nB+BLYgVI
 k60CpiwsZ3G93vUEIO3IyNoH/f0wYK8m9TrDHudUZg3qX4waLG5M43q7Wgc/MbQ
 ITxOUSQv7c7ugFFDzQGBzZswY6786m86gpEIbb3OhjZnzcvQAaRHhdlQWIMm2nr
 AgMBAAEwDQYJKoZIhvcNAQEFBQADgYEAtCu4nUhVVxYUntneD9+h8Mg9q6q+auN
 KyExzyLwaxlAoo7TJHidbtS4J5iNmZgXL0FkbFFBjvSfpJIlJ00zbhNYS5f6Guo
 EDmFJl0ZxBHjJnyp378OD8uTs7fLvjx79LjSTbNYiytVbZPQUQ5Yaxu2jXnimvw
 3rrszlaEXAMPLE=
-----END CERTIFICATE-----",
  "keyPair": {
```

```
    "PublicKey": "-----BEGIN PUBLIC KEY-----\
MIIBIjANBgkqhkEXAMPLEQEFAAOCAQ8AMIIBCgKCAQEAEXAMPLE1nnyJwKSMHw4h\
MMEXAMPLEuuN/dMAS3fyce8DW/4+EXAMPLEyjmoF/YVF/gHr99VEEXAMPLE5VF13\
59VK7cEXAMPLE67GK+y+jikqXOgHh/xJTwo
+sGpWEXAMPLEDz18xOd2ka4tCzuWEXAMPLEahJbYkCPUBSU8opVkR7qkEXAMPLE1DR6sx2HocliOOLtu6Fkw91swQWE
\\GB3ZPrNh0PzQYvjUStZeccyNCx2EXAMPLEvp9mQOUXP6plfgxwKRX2fEXAMPLEDa\
hJLXkX3rHU2xbxJSq7D+XEXAMPLEcw+LyFhI5mgFRl88eGdsAEXAMPLElnI9EesG\
FQIDAQAB\
-----END PUBLIC KEY-----\
",
    "PrivateKey": "-----BEGIN RSA PRIVATE KEY-----\
key omitted for security reasons\
-----END RSA PRIVATE KEY-----\
"
  }
}
```

Save the certificate's Amazon Resource Name (ARN) to use to configure the certificate later.

Create the certificate from a private key in an HSM

> **Note**
>
> This feature is available for v2.5.3 and later of the Greengrass nucleus component. AWS IoT Greengrass doesn't currently support this feature on Windows core devices.

To create the thing certificate

1. On the core device, initialize a PKCS#11 token in the HSM, and generate a private key. The private key must be an RSA key with an RSA-2048 key size (or larger) or an ECC key.

   > **Note**
   >
   > To use a hardware security module with ECC keys, you must use Greengrass nucleus v2.5.6 or later.
   > To use a hardware security module and secret manager, you must use a hardware security module with RSA keys.

Check the documentation for your HSM to learn how to initialize the token and generate the private key. If your HSM supports object IDs, specify an object ID when you generate the private key. Save the slot ID, user PIN, object label, object ID (if your HSM uses one) that you specify when you initialize the token and generate the private key. You use these values later when you import the thing certificate to the HSM and configure the AWS IoT Greengrass Core software.

2. Create a certificate signing request (CSR) from the private key. AWS IoT uses this CSR to create a thing certificate for the private key that you generated in the HSM. For information about how to create a CSR from the private key, see the documentation for your HSM. The CSR is a file, such as `iotdevicekey.csr`.
3. Copy the CSR from the device to your development computer. If SSH and SCP are enabled on the development computer and the device, you can use the `scp` command on your development computer to transfer the CSR. Replace *`device-ip-address`* with the IP address of your device, and replace *`~/iotdevicekey.csr`* with the path to the CSR file on the device.

   ```
   scp device-ip-address:~/iotdevicekey.csr iotdevicekey.csr
   ```

4. On your development computer, create a folder where you download the certificate for the AWS IoT thing.

   ```
   mkdir greengrass-v2-certs
   ```

5. Use the CSR file to create and download the certificate for the AWS IoT thing to your development computer.

   ```
   aws iot create-certificate-from-csr --set-as-active --certificate-signing-
   request=file://iotdevicekey.csr --certificate-pem-outfile greengrass-v2-certs/
   device.pem.crt
   ```

   The response looks similar to the following example, if the request succeeds.

   ```
   {
     "certificateArn": "arn:aws:iot:us-west-2:123456789012:cert/
   aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4",
     "certificateId":
    "aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4",
     "certificatePem": "-----BEGIN CERTIFICATE-----
   ```

```
MIICiTCCAfICCQD6m7oRw0uXOjANBgkqhkiG9w
 0BAQUFADCBiDELMAkGA1UEBhMCVVMxCzAJBgNVBAgTAldBMRAwDgYDVQQHEwdTZ
 WF0dGxlMQ8wDQYDVQQKEwZBbWF6b24xFDASBgNVBAsTC0lBTSBDb25zb2xlMRIw
 EAYDVQQDEwlUZXN0Q2lsYWMxHzAdBgkqhkiG9w0BCQEWEG5vb25lQGFtYXpvbi5
 jb20wHhcNMTEwNDI1MjA0NTIxWhcNMTIwNDI0MjA0NTIxWjCBiDELMAkGA1UEBh
 MCVVMxCzAJBgNVBAgTAldBMRAwDgYDVQQHEwdTZWF0dGxlMQ8wDQYDVQQKEwZBb
 WF6b24xFDASBgNVBAsTC0lBTSBDb25zb2xlMRIwEAYDVQQDEwlUZXN0Q2lsYWMx
 HzAdBgkqhkiG9w0BCQEWEG5vb25lQGFtYXpvbi5jb20wgZ8wDQYJKoZIhvcNAQE
 BBQADgY0AMIGJAoGBAMaK0dn+a4GmWIWJ21uUSfwfEvySWtC2XADZ4nB+BLYgVI
 k60CpiwsZ3G93vUEIO3IyNoH/f0wYK8m9TrDHudUZg3qX4waLG5M43q7Wgc/MbQ
 ITxOUSQv7c7ugFFDzQGBzZswY6786m86gpEIbb3OhjZnzcvQAaRHhdlQWIMm2nr
 AgMBAAEwDQYJKoZIhvcNAQEFBQADgYEAtCu4nUhVVxYUntneD9+h8Mg9q6q+auN
 KyExzyLwaxlAoo7TJHidbtS4J5iNmZgXL0FkbFFBjvSfpJIlJ00zbhNYS5f6Guo
 EDmFJl0ZxBHjJnyp378OD8uTs7fLvjx79LjSTbNYiytVbZPQUQ5Yaxu2jXnimvw
 3rrszlaEXAMPLE=
-----END CERTIFICATE-----"
}
```

Save the certificate's ARN to use to configure the certificate later.

Next, configure the thing certificate. For more information, see Configure the thing certificate.

Create a token exchange role

Greengrass core devices use an IAM service role, called the *token exchange role*, to authorize calls to AWS services. The device uses the AWS IoT credentials provider to get temporary AWS credentials for this role, which allows the device to interact with AWS IoT, send logs to Amazon CloudWatch Logs, and download custom component artifacts from Amazon S3. For more information, see Authorize core devices to interact with AWS services.

You use an AWS IoT *role alias* to configure the token exchange role for Greengrass core devices. Role aliases enable you to change the token exchange role for a device but keep the device configuration the same. For more information, see Authorizing direct calls to AWS services in the *AWS IoT Core Developer Guide*.

In this section, you create a token exchange IAM role and an AWS IoT role alias that points to the role. If you have already set up a Greengrass core device, you can use its token exchange role and role alias instead of creating new ones. Then, you configure your device's AWS IoT thing to use that role and alias.

To create a token exchange IAM role

1. Create an IAM role that your device can use as a token exchange role. Do the following:

   a. Create a file that contains the trust policy document that the token exchange role requires.

      For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

      ```
      nano device-role-trust-policy.json
      ```

      Copy the following JSON into the file.

      ```
      {
        "Version": "2012-10-17",
        "Statement": [
          {
            "Effect": "Allow",
            "Principal": {
              "Service": "credentials.iot.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
          }
        ]
      }
      ```

   b. Create the token exchange role with the trust policy document.

      - Replace *GreengrassV2TokenExchangeRole* with the name of the IAM role to create.

      ```
      aws iam create-role --role-name GreengrassV2TokenExchangeRole --assume-role-
      policy-document file://device-role-trust-policy.json
      ```

      The response looks similar to the following example, if the request succeeds.

      ```
      {
        "Role": {
          "Path": "/",
          "RoleName": "GreengrassV2TokenExchangeRole",
      ```

```
    "RoleId": "AROAZ2YMUHYHK5OKM77FB",
    "Arn": "arn:aws:iam::123456789012:role/GreengrassV2TokenExchangeRole",
    "CreateDate": "2021-02-06T00:13:29+00:00",
    "AssumeRolePolicyDocument": {
      "Version": "2012-10-17",
      "Statement": [
        {
          "Effect": "Allow",
          "Principal": {
            "Service": "credentials.iot.amazonaws.com"
          },
          "Action": "sts:AssumeRole"
        }
      ]
    }
  }
```

c. Create a file that contains the access policy document that the token exchange role requires.

For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

```
nano device-role-access-policy.json
```

Copy the following JSON into the file.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents",
        "logs:DescribeLogStreams",
        "s3:GetBucketLocation"
      ],
      "Resource": "*"
    }
  ]
```

```
}
```

> **Note**
>
> This access policy doesn't allow access to component artifacts in S3 buckets. To deploy custom components that define artifacts in Amazon S3, you must add permissions to the role to allow your core device to retrieve component artifacts. For more information, see Allow access to S3 buckets for component artifacts. If you don't yet have an S3 bucket for component artifacts, you can add these permissions later after you create a bucket.

d. Create the IAM policy from the policy document.

- Replace *GreengrassV2TokenExchangeRoleAccess* with the name of the IAM policy to create.

```
aws iam create-policy --policy-name GreengrassV2TokenExchangeRoleAccess --
policy-document file://device-role-access-policy.json
```

The response looks similar to the following example, if the request succeeds.

```
{
  "Policy": {
    "PolicyName": "GreengrassV2TokenExchangeRoleAccess",
    "PolicyId": "ANPAZ2YMUHYHACI7C5Z66",
    "Arn": "arn:aws:iam::123456789012:policy/
GreengrassV2TokenExchangeRoleAccess",
    "Path": "/",
    "DefaultVersionId": "v1",
    "AttachmentCount": 0,
    "PermissionsBoundaryUsageCount": 0,
    "IsAttachable": true,
    "CreateDate": "2021-02-06T00:37:17+00:00",
    "UpdateDate": "2021-02-06T00:37:17+00:00"
  }
}
```

e. Attach the IAM policy to the token exchange role.

- Replace *GreengrassV2TokenExchangeRole* with the name of the IAM role.

- Replace the policy ARN with the ARN of the IAM policy that you created in the previous step.

```
aws iam attach-role-policy --role-name GreengrassV2TokenExchangeRole --policy-
arn arn:aws:iam::123456789012:policy/GreengrassV2TokenExchangeRoleAccess
```

The command doesn't have any output if the request succeeds.

2. Create an AWS IoT role alias that points to the token exchange role.

- Replace *GreengrassCoreTokenExchangeRoleAlias* with the name of the role alias to create.
- Replace the role ARN with the ARN of the IAM role that you created in the previous step.

```
aws iot create-role-alias --role-alias GreengrassCoreTokenExchangeRoleAlias --role-
arn arn:aws:iam::123456789012:role/GreengrassV2TokenExchangeRole
```

The response looks similar to the following example, if the request succeeds.

```
{
  "roleAlias": "GreengrassCoreTokenExchangeRoleAlias",
  "roleAliasArn": "arn:aws:iot:us-west-2:123456789012:rolealias/
GreengrassCoreTokenExchangeRoleAlias"
}
```

Note

To create a role alias, you must have permission to pass the token exchange IAM role to AWS IoT. If you receive an error message when you try to create a role alias, check that your AWS user has this permission. For more information, see Granting a user permissions to pass a role to an AWS service in the *AWS Identity and Access Management User Guide*.

3. Create and attach an AWS IoT policy that allows your Greengrass core device to use the role alias to assume the token exchange role. If you have set up a Greengrass core device before, you can attach its role alias AWS IoT policy instead of creating a new one. Do the following:

a. (Optional) Create a file that contains the AWS IoT policy document that the role alias requires.

   For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

   ```
   nano greengrass-v2-iot-role-alias-policy.json
   ```

   Copy the following JSON into the file.

   - Replace the resource ARN with the ARN of your role alias.

   ```
   {
     "Version":"2012-10-17",
     "Statement": [
       {
         "Effect": "Allow",
         "Action": "iot:AssumeRoleWithCertificate",
         "Resource": "arn:aws:iot:us-west-2:123456789012:rolealias/
   GreengrassCoreTokenExchangeRoleAlias"
       }
     ]
   }
   ```

b. Create an AWS IoT policy from the policy document.

   - Replace *GreengrassCoreTokenExchangeRoleAliasPolicy* with the name of the AWS IoT policy to create.

   ```
   aws iot create-policy --policy-name GreengrassCoreTokenExchangeRoleAliasPolicy
    --policy-document file://greengrass-v2-iot-role-alias-policy.json
   ```

   The response looks similar to the following example, if the request succeeds.

   ```
   {
     "policyName": "GreengrassCoreTokenExchangeRoleAliasPolicy",
     "policyArn": "arn:aws:iot:us-west-2:123456789012:policy/
   GreengrassCoreTokenExchangeRoleAliasPolicy",
   ```

```
  "policyDocument": "{
    \\"Version\\":\\"2012-10-17       \\",
    \\"Statement\\": [
      {
        \\"Effect\\": \\"Allow\\",
        \\"Action\\": \\"iot:AssumeRoleWithCertificate\\",
        \\"Resource\\": \\"arn:aws:iot:us-west-2:123456789012:rolealias/
GreengrassCoreTokenExchangeRoleAlias\\"
      }
    ]
  }",
  "policyVersionId": "1"
}
```

c. Attach the AWS IoT policy to the AWS IoT thing's certificate.

- Replace *GreengrassCoreTokenExchangeRoleAliasPolicy* with the name of the role alias AWS IoT policy.
- Replace the target ARN with the ARN of the certificate for your AWS IoT thing.

```
aws iot attach-policy --policy-name GreengrassCoreTokenExchangeRoleAliasPolicy
 --target arn:aws:iot:us-west-2:123456789012:cert/
aa0b7958770878eabe251d8a7ddd547f4889c524c9b574ab9fbf65f32248b1d4
```

The command doesn't have any output if the request succeeds.

Download certificates to the device

Earlier, you downloaded your device's certificate to your development computer. In this section, you copy the certificate to your core device to set up the device with the certificates that it uses to connect to AWS IoT. You also download the Amazon root certificate authority (CA) certificate. If you use an HSM, you also import the certificate file into the HSM in this section.

- If you created the thing certificate and private key in the AWS IoT service earlier, follow the steps to download the certificates with private key and certificate files.
- If you created the thing certificate from a private key in a hardware security module (HSM) earlier, follow the steps to download the certificates with the private key and certificate in an HSM.

Download certificates with private key and certificate files

To download certificates to the device

1. Copy the AWS IoT thing certificate from your development computer to the device. If SSH and SCP are enabled on the development computer and the device, you can use the `scp` command on your development computer to transfer the certificate. Replace *`device-ip-address`* with the IP address of your device.

   ```
   scp -r greengrass-v2-certs/ device-ip-address:~
   ```

2. Create the Greengrass root folder on the device. You'll later install the AWS IoT Greengrass Core software to this folder.

   > **Note**
   >
   > Windows has a path length limitation of 260 characters. If you are using Windows, use a root folder like `C:\greengrass\v2` or `D:\greengrass\v2` to keep the Greengrass components paths under the 260 character limit.

   Linux or Unix

   - Replace */greengrass/v2* with the folder to use.

   ```
   sudo mkdir -p /greengrass/v2
   ```

   Windows Command Prompt

   - Replace *C:\greengrass\v2* with the folder to use.

   ```
   mkdir C:\greengrass\v2
   ```

   PowerShell

   - Replace *C:\greengrass\v2* with the folder to use.

```
mkdir C:\greengrass\v2
```

3. (Linux only) Set the permissions of the parent of the Greengrass root folder.

   - Replace */greengrass* with the parent of the root folder.

   ```
   sudo chmod 755 /greengrass
   ```

4. Copy the AWS IoT thing certificates to the Greengrass root folder.

   Linux or Unix

   - Replace */greengrass/v2* with the Greengrass root folder.

   ```
   sudo cp -R ~/greengrass-v2-certs/* /greengrass/v2
   ```

   Windows Command Prompt

   - Replace *C:\greengrass\v2* with the folder to use.

   ```
   robocopy %USERPROFILE%\greengrass-v2-certs C:\greengrass\v2 /E
   ```

   PowerShell

   - Replace *C:\greengrass\v2* with the folder to use.

   ```
   cp -Path ~\greengrass-v2-certs\* -Destination C:\greengrass\v2
   ```

5. Download the Amazon root certificate authority (CA) certificate. AWS IoT certificates are associated with Amazon's root CA certificate by default. Download the CA1 certificate and the CA3 certificate.

   Linux or Unix

   - Replace */greengrass/v2* or *C:\greengrass\v2* with the Greengrass root folder.

```
sudo curl -o /greengrass/v2/AmazonRootCA1.pem https://www.amazontrust.com/
repository/AmazonRootCA1.pem
sudo curl -o - https://www.amazontrust.com/repository/AmazonRootCA3.pem >> /
greengrass/v2/AmazonRootCA1.pem
```

Windows Command Prompt (CMD)

```
curl -o C:\greengrass\v2\\AmazonRootCA1.pem https://www.amazontrust.com/
repository/AmazonRootCA1.pem
```

PowerShell

```
iwr -Uri https://www.amazontrust.com/repository/AmazonRootCA1.pem -OutFile C:
\greengrass\v2\\AmazonRootCA1.pem
```

Download certificates with the private key and certificate in an HSM

> **Note**
>
> This feature is available for v2.5.3 and later of the Greengrass nucleus component. AWS IoT Greengrass doesn't currently support this feature on Windows core devices.

To download certificates to the device

1. Copy the AWS IoT thing certificate from your development computer to the device. If SSH and SCP are enabled on the development computer and the device, you can use the `scp` command on your development computer to transfer the certificate. Replace *`device-ip-address`* with the IP address of your device.

   ```
   scp -r greengrass-v2-certs/ device-ip-address:~
   ```

2. Create the Greengrass root folder on the device. You'll later install the AWS IoT Greengrass Core software to this folder.

> **Note**
>
> Windows has a path length limitation of 260 characters. If you are using Windows, use a root folder like `C:\greengrass\v2` or `D:\greengrass\v2` to keep the Greengrass components paths under the 260 character limit.

Linux or Unix

- Replace */greengrass/v2* with the folder to use.

```
sudo mkdir -p /greengrass/v2
```

Windows Command Prompt

- Replace *C:\greengrass\v2* with the folder to use.

```
mkdir C:\greengrass\v2
```

PowerShell

- Replace *C:\greengrass\v2* with the folder to use.

```
mkdir C:\greengrass\v2
```

3. (Linux only) Set the permissions of the parent of the Greengrass root folder.

   - Replace */greengrass* with the parent of the root folder.

   ```
   sudo chmod 755 /greengrass
   ```

4. Import the thing certificate file, `~/greengrass-v2-certs/device.pem.crt`, into the HSM. Check the documentation for your HSM to learn how to import certificates into it. Import the certificate using the same token, slot ID, user PIN, object label, and object ID (if your HSM uses one) where you generated the private key in the HSM earlier.

> **Note**
>
> If you generated the private key earlier without an object ID, and the certificate has an object ID, set the private key's object ID to the same value as the certificate. Check the documentation for your HSM to learn how to set the object ID for the private key object.

5. (Optional) Delete the thing certificate file, so that it exists only in the HSM.

```
rm ~/greengrass-v2-certs/device.pem.crt
```

6. Download the Amazon root certificate authority (CA) certificate. AWS IoT certificates are associated with Amazon's root CA certificate by default. Download both the CA1 and the CA3 certificate.

   Linux or Unix

   - Replace */greengrass/v2* or *C:\greengrass\v2* with the Greengrass root folder.

```
sudo curl -o /greengrass/v2/AmazonRootCA1.pem https://www.amazontrust.com/
repository/AmazonRootCA1.pem
sudo curl -o - https://www.amazontrust.com/repository/AmazonRootCA3.pem >> /
greengrass/v2/AmazonRootCA1.pem
```

   Windows Command Prompt (CMD)

```
curl -o C:\greengrass\v2\\AmazonRootCA1.pem https://www.amazontrust.com/
repository/AmazonRootCA1.pem
```

   PowerShell

```
iwr -Uri https://www.amazontrust.com/repository/AmazonRootCA1.pem -OutFile C:
\greengrass\v2\\AmazonRootCA1.pem
```

Set up the device environment

Follow the steps in this section to set up a Linux or Windows device to use as your AWS IoT Greengrass core device.

Set up a Linux device

To set up a Linux device for AWS IoT Greengrass V2

1. Install the Java runtime, which AWS IoT Greengrass Core software requires to run. We recommend that you use Amazon Corretto or OpenJDK long-term support versions. Version 8 or higher is required. The following commands show you how to install OpenJDK on your device.

   - For Debian-based or Ubuntu-based distributions:

     ```
     sudo apt install default-jdk
     ```

   - For Red Hat-based distributions:

     ```
     sudo yum install java-11-openjdk-devel
     ```

   - For Amazon Linux 2:

     ```
     sudo amazon-linux-extras install java-openjdk11
     ```

   - For Amazon Linux 2023:

     ```
     sudo dnf install java-11-amazon-corretto -y
     ```

   When the installation completes, run the following command to verify that Java runs on your Linux device.

   ```
   java -version
   ```

   The command prints the version of Java that runs on the device. For example, on a Debian-based distribution, the output might look similar to the following sample.

   ```
   openjdk version "11.0.9.1" 2020-11-04
   ```

```
OpenJDK Runtime Environment (build 11.0.9.1+1-post-Debian-1deb10u2)
OpenJDK 64-Bit Server VM (build 11.0.9.1+1-post-Debian-1deb10u2, mixed mode)
```

2. (Optional) Create the default system user and group that runs components on the device. You can also choose to let the AWS IoT Greengrass Core software installer create this user and group during installation with the `--component-default-user` installer argument. For more information, see Installer arguments.

   ```
   sudo useradd --system --create-home ggc_user
   sudo groupadd --system ggc_group
   ```

3. Verify that the user that runs the AWS IoT Greengrass Core software (typically `root`), has permission to run `sudo` with any user and any group.

   a. Run the following command to open the `/etc/sudoers` file.

      ```
      sudo visudo
      ```

   b. Verify that the permission for the user looks like the following example.

      ```
      root    ALL=(ALL:ALL) ALL
      ```

4. (Optional) To run containerized Lambda functions, you must enable cgroups v1, and you must enable and mount the *memory* and *devices* cgroups. If you don't plan to run containerized Lambda functions, you can skip this step.

   To enable these cgroups options, boot the device with the following Linux kernel parameters.

   ```
   cgroup_enable=memory cgroup_memory=1 systemd.unified_cgroup_hierarchy=0
   ```

   For information about viewing and setting kernel parameters for your device, see the documentation for your operating system and boot loader. Follow the instructions to permanently set the kernel parameters.

5. Install all other required dependencies on your device as indicated by the list of requirements in Device requirements.

Set up a Windows device

> **Note**
>
> This feature is available for v2.5.0 and later of the Greengrass nucleus component.

To set up a Windows device for AWS IoT Greengrass V2

1. Install the Java runtime, which AWS IoT Greengrass Core software requires to run. We recommend that you use Amazon Corretto or OpenJDK long-term support versions. Version 8 or higher is required.
2. Check whether Java is available on the PATH system variable, and add it if not. The LocalSystem account runs the AWS IoT Greengrass Core software, so you must add Java to the PATH system variable instead of the PATH user variable for your user. Do the following:
   a. Press the Windows key to open the start menu.
   b. Type **`environment variables`** to search for the system options from the start menu.
   c. In the start menu search results, choose **Edit the system environment variables** to open the **System properties** window.
   d. Choose **Environment variables...** to open the **Environment Variables** window.
   e. Under **System variables**, select **Path**, and then choose **Edit**. In the **Edit environment variable** window, you can view each path on a separate line.
   f. Check if the path to the Java installation's `bin` folder is present. The path might look similar to the following example.

      ```
      C:\\Program Files\\Amazon Corretto\\jdk11.0.13_8\\bin
      ```

   g. If the Java installation's `bin` folder is missing from **Path**, choose **New** to add it, and then choose **OK**.
3. Open the Windows Command Prompt (`cmd.exe`) as an administrator.
4. Create the default user in the LocalSystem account on the Windows device. Replace *`password`* with a secure password.

   ```
   net user /add ggc_user password
   ```

Tip

Depending on your Windows configuration, the user's password might be set to expire at a date in the future. To ensure your Greengrass applications continue to operate, track when the password expires, and update it before it expires. You can also set the user's password to never expire.

- To check when a user and its password expire, run the following command.

```
net user ggc_user | findstr /C:expires
```

- To set a user's password to never expire, run the following command.

```
wmic UserAccount where "Name='ggc_user'" set PasswordExpires=False
```

- If you're using Windows 10 or later where the wmic command is deprecated, run the following PowerShell command.

```
Get-CimInstance -Query "SELECT * from Win32_UserAccount WHERE name =
 'ggc_user'" | Set-CimInstance -Property @{PasswordExpires="False"}
```

5. Download and install the PsExec utility from Microsoft on the device.
6. Use the PsExec utility to store the user name and password for the default user in the Credential Manager instance for the LocalSystem account. Replace *password* with the user's password that you set earlier.

```
psexec -s cmd /c cmdkey /generic:ggc_user /user:ggc_user /pass:password
```

If the **PsExec License Agreement** opens, choose **Accept** to agree to the license and run the command.

Note

On Windows devices, the LocalSystem account runs the Greengrass nucleus, and you must use the PsExec utility to store the default user information in the LocalSystem account. Using the Credential Manager application stores this information in the Windows account of the currently logged on user, instead of the LocalSystem account.

Download the AWS IoT Greengrass Core software

You can download the latest version of the AWS IoT Greengrass Core software from the following location:

- https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip

> **Note**
>
> You can download a specific version of the AWS IoT Greengrass Core software from the following location. Replace *version* with the version to download.
>
> ```
> https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-version.zip
> ```

To download the AWS IoT Greengrass Core software

1. On your core device, download the AWS IoT Greengrass Core software to a file named `greengrass-nucleus-latest.zip`.

   Linux or Unix

   ```
   curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-
   latest.zip > greengrass-nucleus-latest.zip
   ```

   Windows Command Prompt (CMD)

   ```
   curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-
   latest.zip > greengrass-nucleus-latest.zip
   ```

   PowerShell

   ```
   iwr -Uri https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-
   latest.zip -OutFile greengrass-nucleus-latest.zip
   ```

   By downloading this software, you agree to the Greengrass Core Software License Agreement.

2. (Optional) To verify the Greengrass nucleus software signature

> **Note**
>
> This feature is available with Greengrass nucleus version 2.9.5 and later.

a. Use the following command to verify your Greengrass nucleus artifact's signature:

   Linux or Unix

   ```
   jarsigner -verify -certs -verbose greengrass-nucleus-latest.zip
   ```

   Windows Command Prompt (CMD)

   The file name might look different depending on the JDK version you install. Replace *jdk17.0.6_10* with the JDK version you installed.

   ```
   "C:\\Program Files\\Amazon Corretto\\jdk17.0.6_10\\bin\\jarsigner.exe" -
   verify -certs -verbose greengrass-nucleus-latest.zip
   ```

   PowerShell

   The file name might look different depending on the JDK version you install. Replace *jdk17.0.6_10* with the JDK version you installed.

   ```
   'C:\\Program Files\\Amazon Corretto\\jdk17.0.6_10\\bin\\jarsigner.exe' -
   verify -certs -verbose greengrass-nucleus-latest.zip
   ```

b. The `jarsigner` invocation yields output that indicates the results of the verification.

   i. If the Greengrass nucleus zip file is signed, the output contains the following statement:

   ```
   jar verified.
   ```

   ii. If the Greengrass nucleus zip file isn't signed, the output contains the following statement:

   ```
   jar is unsigned.
   ```

c. If you provided the Jarsigner `-certs` option along with `-verify` and `-verbose` options, the output also includes detailed signer certificate information.

3. Unzip the AWS IoT Greengrass Core software to a folder on your device. Replace *GreengrassInstaller* with the folder that you want to use.

   Linux or Unix

   ```
   unzip greengrass-nucleus-latest.zip -d GreengrassInstaller && rm greengrass-nucleus-latest.zip
   ```

   Windows Command Prompt (CMD)

   ```
   mkdir GreengrassInstaller && tar -xf greengrass-nucleus-latest.zip -C GreengrassInstaller && del greengrass-nucleus-latest.zip
   ```

   PowerShell

   ```
   Expand-Archive -Path greengrass-nucleus-latest.zip -DestinationPath .\GreengrassInstaller
   rm greengrass-nucleus-latest.zip
   ```

4. (Optional) Run the following command to see the version of the AWS IoT Greengrass Core software.

   ```
   java -jar ./GreengrassInstaller/lib/Greengrass.jar --version
   ```

Important

If you install a version of the Greengrass nucleus earlier than v2.4.0, don't remove this folder after you install the AWS IoT Greengrass Core software. The AWS IoT Greengrass Core software uses the files in this folder to run.
If you downloaded the latest version of the software, you install v2.4.0 or later, and you can remove this folder after you install the AWS IoT Greengrass Core software.

Install the AWS IoT Greengrass Core software

Run the installer with arguments that specify the following actions:

- Install from a partial configuration file that specifies to use the AWS resources and certificates that you created earlier. The AWS IoT Greengrass Core software uses a configuration file that specifies the configuration of every Greengrass component on the device. The installer creates a complete configuration file from the partial configuration file that you provide.
- Specify to use the `ggc_user` system user to run software components on the core device. On Linux devices, this command also specifies to use the `ggc_group` system group, and the installer creates the system user and group for you.
- Set up the AWS IoT Greengrass Core software as a system service that runs at boot. On Linux devices, this requires the Systemd init system.

> **Important**
>
> On Windows core devices, you must set up the AWS IoT Greengrass Core software as a system service.

For more information about the arguments that you can specify, see Installer arguments.

> **Note**
>
> If you are running AWS IoT Greengrass on a device with limited memory, you can control the amount of memory that AWS IoT Greengrass Core software uses. To control memory allocation, you can set JVM heap size options in the `jvmOptions` configuration parameter in your nucleus component. For more information, see Control memory allocation with JVM options.

- If you created the thing certificate and private key in the AWS IoT service earlier, follow the steps to install the AWS IoT Greengrass Core software with private key and certificate files.
- If you created the thing certificate from a private key in a hardware security module (HSM) earlier, follow the steps to install the AWS IoT Greengrass Core software with the private key and certificate in an HSM.

Install the AWS IoT Greengrass Core software with private key and certificate files

To install the AWS IoT Greengrass Core software

1. Check the version of the AWS IoT Greengrass Core software.

   - Replace *GreengrassInstaller* with the path to the folder that contains the software.

   ```
   java -jar ./GreengrassInstaller/lib/Greengrass.jar --version
   ```

2. Use a text editor to create a configuration file named `config.yaml` to provide to the installer.

   For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

   ```
   nano GreengrassInstaller/config.yaml
   ```

   Copy the following YAML content into the file. This partial configuration file specifies system parameters and Greengrass nucleus parameters.

   ```
   ---
   system:
     certificateFilePath: "/greengrass/v2/device.pem.crt"
     privateKeyPath: "/greengrass/v2/private.pem.key"
     rootCaPath: "/greengrass/v2/AmazonRootCA1.pem"
     rootpath: "/greengrass/v2"
     thingName: "MyGreengrassCore"
   services:
     aws.greengrass.Nucleus:
       componentType: "NUCLEUS"
       version: "2.18.4"
       configuration:
         awsRegion: "us-west-2"
         iotRoleAlias: "GreengrassCoreTokenExchangeRoleAlias"
         fipsMode: "true"
         iotDataEndpoint: "data.iot-fips.us-west-2.amazonaws.com"
         greengrassDataPlaneEndpoint: "iotData"
         iotCredEndpoint: "data.credentials.iot-fips.us-west-2.amazonaws.com"
   ```

   Then, do the following:

- Replace each instance of */greengrass/v2* with the Greengrass root folder.
- Replace *MyGreengrassCore* with the name of the AWS IoT thing.
- Replace *2.18.4* with the version of the AWS IoT Greengrass Core software.
- Replace *us-west-2* with the AWS Region where you created the resources.
- Replace *GreengrassCoreTokenExchangeRoleAlias* with the name of the token exchange role alias.
- Replace the *iotDataEndpoint* with your AWS IoT data endpoint.
- Replace the *iotCredEndpoint* with your AWS IoT credentials endpoint.

3. Run the installer, and specify `--init-config` to provide the configuration file.

- Replace */greengrass/v2* or *C:\greengrass\v2* with the Greengrass root folder.
- Replace each instance of *GreengrassInstaller* with the folder where you unpacked the installer.

Linux or Unix

```
sudo -E java -Droot="/greengrass/v2" -Dlog.store=FILE \
  -jar ./GreengrassInstaller/lib/Greengrass.jar \
  --init-config ./GreengrassInstaller/config.yaml \
  --component-default-user ggc_user:ggc_group \
  --setup-system-service true
```

Windows Command Prompt (CMD)

```
java -Droot="C:\greengrass\v2" "-Dlog.store=FILE" ^
  -jar ./GreengrassInstaller/lib/Greengrass.jar ^
  --init-config ./GreengrassInstaller/config.yaml ^
  --component-default-user ggc_user ^
  --setup-system-service true
```

PowerShell

```
java -Droot="C:\greengrass\v2" "-Dlog.store=FILE" `
  -jar ./GreengrassInstaller/lib/Greengrass.jar `
  --init-config ./GreengrassInstaller/config.yaml `
  --component-default-user ggc_user `
```

```
  --setup-system-service true
```

> **Important**
>
> On Windows core devices, you must specify `--setup-system-service true` to set up the AWS IoT Greengrass Core software as a system service.

If you specify `--setup-system-service true`, the installer prints `Successfully set up Nucleus as a system service` if it set up and ran the software as a system service. Otherwise, the installer doesn't output any message if it installs the software successfully.

> **Note**
>
> You can't use the `deploy-dev-tools` argument to deploy local development tools when you run the installer without the `--provision true` argument. For information about deploying the Greengrass CLI directly on your device, see Greengrass Command Line Interface.

4. Verify the installation by viewing the files in the root folder.

   Linux or Unix

   ```
   ls /greengrass/v2
   ```

   Windows Command Prompt (CMD)

   ```
   dir C:\greengrass\v2
   ```

   PowerShell

   ```
   ls C:\greengrass\v2
   ```

   If the installation succeeded, the root folder contains several folders, such as `config`, `packages`, and `logs`.

Install the AWS IoT Greengrass Core software with the private key and certificate in an HSM

> **Note**
>
> This feature is available for v2.5.3 and later of the Greengrass nucleus component. AWS IoT Greengrass doesn't currently support this feature on Windows core devices.

To install the AWS IoT Greengrass Core software

1. Check the version of the AWS IoT Greengrass Core software.

   - Replace *GreengrassInstaller* with the path to the folder that contains the software.

   ```
   java -jar ./GreengrassInstaller/lib/Greengrass.jar --version
   ```

2. To enable the AWS IoT Greengrass Core software to use the private key and certificate in the HSM, install the PKCS#11 provider component when you install the AWS IoT Greengrass Core software. The PKCS#11 provider component is a plugin that you can configure during installation. You can download the latest version of the PKCS#11 provider component from the following location:

   - https://d2s8p88vqu9w66.cloudfront.net/releases/Pkcs11Provider/aws.greengrass.crypto.Pkcs11Provider-latest.jar

   Download the PKCS#11 provider plugin to a file named `aws.greengrass.crypto.Pkcs11Provider.jar`. Replace *GreengrassInstaller* with the folder that you want to use.

   ```
   curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/Pkcs11Provider/
   aws.greengrass.crypto.Pkcs11Provider-latest.jar > GreengrassInstaller/
   aws.greengrass.crypto.Pkcs11Provider.jar
   ```

   By downloading this software, you agree to the Greengrass Core Software License Agreement.

3. Use a text editor to create a configuration file named `config.yaml` to provide to the installer.

   For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

```
nano GreengrassInstaller/config.yaml
```

Copy the following YAML content into the file. This partial configuration file specifies system parameters, Greengrass nucleus parameters, and PKCS#11 provider parameters.

```
---
system:
  certificateFilePath: "/greengrass/v2/device.pem.crt"
  privateKeyPath: "/greengrass/v2/private.pem.key"
  rootCaPath: "/greengrass/v2/AmazonRootCA1.pem"
  rootpath: "/greengrass/v2"
  thingName: "MyGreengrassCore"
services:
  aws.greengrass.Nucleus:
    componentType: "NUCLEUS"
    version: "2.18.4"
    configuration:
      awsRegion: "us-west-2"
      iotRoleAlias: "GreengrassCoreTokenExchangeRoleAlias"
      fipsMode: "true"
      iotDataEndpoint: "data.iot-fips.us-west-2.amazonaws.com"
      greengrassDataPlaneEndpoint: "iotData"
      iotCredEndpoint: "data.credentials.iot-fips.us-west-2.amazonaws.com"
```

Then, do the following:

- Replace each instance of *iotdevicekey* in the PKCS#11 URIs with the object label where you created the private key and imported the certificate.
- Replace each instance of */greengrass/v2* with the Greengrass root folder.
- Replace *MyGreengrassCore* with the name of the AWS IoT thing.
- Replace *2.18.4* with the version of the AWS IoT Greengrass Core software.
- Replace *us-west-2* with the AWS Region where you created the resources.
- Replace *GreengrassCoreTokenExchangeRoleAlias* with the name of the token exchange role alias.
- Replace the `iotDataEndpoint` with your AWS IoT data endpoint.
- Replace the `iotCredEndpoint` with your AWS IoT credentials endpoint.

- Replace the configuration parameters for the `aws.greengrass.crypto.Pkcs11Provider` component with the values for the HSM configuration on the core device.

4. Run the installer, and specify `--init-config` to provide the configuration file.

   - Replace */greengrass/v2* with the Greengrass root folder.
   - Replace each instance of *GreengrassInstaller* with the folder where you unpacked the installer.

```
sudo -E java -Droot="/greengrass/v2" -Dlog.store=FILE \
  -jar ./GreengrassInstaller/lib/Greengrass.jar \
  --trusted-plugin ./GreengrassInstaller/aws.greengrass.crypto.Pkcs11Provider.jar \
  --init-config ./GreengrassInstaller/config.yaml \
  --component-default-user ggc_user:ggc_group \
  --setup-system-service true
```

   Important

   On Windows core devices, you must specify `--setup-system-service true` to set up the AWS IoT Greengrass Core software as a system service.

   If you specify `--setup-system-service true`, the installer prints `Successfully set up Nucleus as a system service` if it set up and ran the software as a system service. Otherwise, the installer doesn't output any message if it installs the software successfully.

   Note

   You can't use the `deploy-dev-tools` argument to deploy local development tools when you run the installer without the `--provision true` argument. For information about deploying the Greengrass CLI directly on your device, see Greengrass Command Line Interface.

5. Verify the installation by viewing the files in the root folder.

Linux or Unix

```
ls /greengrass/v2
```

Windows Command Prompt (CMD)

```
dir C:\greengrass\v2
```

PowerShell

```
ls C:\greengrass\v2
```

If the installation succeeded, the root folder contains several folders, such as `config`, `packages`, and `logs`.

If you installed the AWS IoT Greengrass Core software as a system service, the installer runs the software for you. Otherwise, you must run the software manually. For more information, see Run the AWS IoT Greengrass Core software.

For more information about how to configure and use the software and AWS IoT Greengrass, see the following:

- Configure the AWS IoT Greengrass Core software
- Develop AWS IoT Greengrass components
- Deploy AWS IoT Greengrass components to devices
- Greengrass Command Line Interface

Install FIPS endpoints with fleet provisioning

This feature is available for v2.4.0 and later of the Greengrass nucleus component.

Install FIPS endpoints on your AWS IoT Greengrass Core software with AWS IoT fleet provisioning for your core devices.

> **Note**
>
> The fleet provisioning plugin for Greengrass nucleus doesn't currently support storing private key and certificate files in a hardware security module (HSM). To use an HSM, install the AWS IoT Greengrass Core software with manual provisioning.

To install the AWS IoT Greengrass Core software with AWS IoT fleet provisioning, you must set up resources in your AWS account that AWS IoT uses to provision Greengrass core devices. These resources include a provisioning template, claim certificates, and a token exchange IAM role. After you create these resources, you can reuse them to provision multiple core devices in a fleet. For more information, see Set up AWS IoT fleet provisioning for Greengrass core devices.

> **Important**
>
> Before you download the AWS IoT Greengrass Core software, check that your core device meets the requirements to install and run the AWS IoT Greengrass Core software v2.0.

Topics

- Prerequisites
- Retrieve AWS IoT endpoints
- Download certificates to the device
- Set up the device environment
- Download the AWS IoT Greengrass Core software
- Download the AWS IoT fleet provisioning plugin
- Install the AWS IoT Greengrass Core software

Prerequisites

To install the AWS IoT Greengrass Core software with AWS IoT fleet provisioning, you must first set up AWS IoT fleet provisioning for Greengrass core devices. After you complete these steps once, you can use fleet provisioning to install the AWS IoT Greengrass Core software on any number of devices.

Retrieve AWS IoT endpoints

Get the FIPS endpoints for your AWS account, and save them to use later. Your device uses these endpoints to connect to AWS IoT. Do the following:

1. Get the FIPS data endpoint for your region in the AWS IoT Core FIPS data plane endpoints. The FIPS data endpoint for your AWS account should look like this: *`data.iot-fips.us-west-2.amazonaws.com`*
2. Get the FIPS credentials endpoint for your region in the AWS IoT Core FIPS data plane endpoints. The FIPS credentials endpoint for your AWS account should look like this: *`data.credentials.iot-fips.us-west-2.amazonaws.com`*

Download certificates to the device

The device uses a claim certificate and private key to authenticate its request to provision AWS resources and acquire an X.509 device certificate. You can embed the claim certificate and private key into the device during manufacturing, or copy the certificate and key to the device during installation. In this section, you copy the claim certificate and private key to the device. You also download the Amazon Root certificate authority (CA) certificate to the device.

Important

Provisioning claim private keys should be secured at all times, including on Greengrass core devices. We recommend that you use Amazon CloudWatch metrics and logs to monitor for indications of misuse, such as unauthorized use of the claim certificate to provision devices. If you detect misuse, disable the provisioning claim certificate so that it can't be used for device provisioning. For more information, see Monitoring AWS IoT in the *AWS IoT Core Developer Guide*.

To help you better manage the number of devices, and which devices, that register themselves in your AWS account, you can specify a pre-provisioning hook when you create a fleet provisioning template. A pre-provisioning hook is an AWS Lambda function that validates template parameters that devices provide during registration. For example, you might create a pre-provisioning hook that checks a device ID against a database to verify that the device has permission to provision. For more information, see Pre-provisioning hooks in the *AWS IoT Core Developer Guide*.

To download claim certificates to the device

1. Copy the claim certificate and private key to the device. If SSH and SCP are enabled on the development computer and the device, you can use the `scp` command on your development computer to transfer the claim certificate and private key. The following example command transfers these files a folder named `claim-certs` on your development computer to the device. Replace *`device-ip-address`* with the IP address of your device.

   ```
   scp -r claim-certs/ device-ip-address:~
   ```

2. Create the Greengrass root folder on the device. You'll later install the AWS IoT Greengrass Core software to this folder.

   > **Note**
   >
   > Windows has a path length limitation of 260 characters. If you are using Windows, use a root folder like `C:\greengrass\v2` or `D:\greengrass\v2` to keep the Greengrass components paths under the 260 character limit.

   Linux or Unix

   - Replace */greengrass/v2* with the folder to use.

   ```
   sudo mkdir -p /greengrass/v2
   ```

   Windows Command Prompt

   - Replace *C:\greengrass\v2* with the folder to use.

   ```
   mkdir C:\greengrass\v2
   ```

   PowerShell

   - Replace *C:\greengrass\v2* with the folder to use.

```
mkdir C:\greengrass\v2
```

3. (Linux only) Set the permissions of the parent of the Greengrass root folder.

   - Replace */greengrass* with the parent of the root folder.

   ```
   sudo chmod 755 /greengrass
   ```

4. Move the claim certificates to the Greengrass root folder.

   - Replace */greengrass/v2* or *C:\greengrass\v2* with the Greengrass root folder.

   Linux or Unix

   ```
   sudo mv ~/claim-certs /greengrass/v2
   ```

   Windows Command Prompt (CMD)

   ```
   move %USERPROFILE%\claim-certs C:\greengrass\v2
   ```

   PowerShell

   ```
   mv -Path ~\claim-certs -Destination C:\greengrass\v2
   ```

5. Download both the CA1 certificate and the CA3 certificate.

   Linux or Unix

   ```
   sudo curl -o - https://www.amazontrust.com/repository/AmazonRootCA3.pem >> /
   greengrass/v2/AmazonRootCA1.pem
   ```

   Windows Command Prompt (CMD)

   ```
   curl -o C:\greengrass\v2\\AmazonRootCA1.pem https://www.amazontrust.com/
   repository/AmazonRootCA1.pem
   ```

PowerShell

```
iwr -Uri https://www.amazontrust.com/repository/AmazonRootCA1.pem -OutFile C:
\greengrass\v2\\AmazonRootCA1.pem
```

Set up the device environment

Follow the steps in this section to set up a Linux or Windows device to use as your AWS IoT Greengrass core device.

Set up a Linux device

To set up a Linux device for AWS IoT Greengrass V2

1. Install the Java runtime, which AWS IoT Greengrass Core software requires to run. We recommend that you use Amazon Corretto or OpenJDK long-term support versions. Version 8 or higher is required. The following commands show you how to install OpenJDK on your device.

   - For Debian-based or Ubuntu-based distributions:

     ```
     sudo apt install default-jdk
     ```

   - For Red Hat-based distributions:

     ```
     sudo yum install java-11-openjdk-devel
     ```

   - For Amazon Linux 2:

     ```
     sudo amazon-linux-extras install java-openjdk11
     ```

   - For Amazon Linux 2023:

     ```
     sudo dnf install java-11-amazon-corretto -y
     ```

   When the installation completes, run the following command to verify that Java runs on your Linux device.

```
java -version
```

The command prints the version of Java that runs on the device. For example, on a Debian-based distribution, the output might look similar to the following sample.

```
openjdk version "11.0.9.1" 2020-11-04
OpenJDK Runtime Environment (build 11.0.9.1+1-post-Debian-1deb10u2)
OpenJDK 64-Bit Server VM (build 11.0.9.1+1-post-Debian-1deb10u2, mixed mode)
```

2. (Optional) Create the default system user and group that runs components on the device. You can also choose to let the AWS IoT Greengrass Core software installer create this user and group during installation with the `--component-default-user` installer argument. For more information, see Installer arguments.

```
sudo useradd --system --create-home ggc_user
sudo groupadd --system ggc_group
```

3. Verify that the user that runs the AWS IoT Greengrass Core software (typically `root`), has permission to run `sudo` with any user and any group.

   a. Run the following command to open the `/etc/sudoers` file.

   ```
   sudo visudo
   ```

   b. Verify that the permission for the user looks like the following example.

   ```
   root    ALL=(ALL:ALL) ALL
   ```

4. (Optional) To run containerized Lambda functions, you must enable cgroups v1, and you must enable and mount the *memory* and *devices* cgroups. If you don't plan to run containerized Lambda functions, you can skip this step.

   To enable these cgroups options, boot the device with the following Linux kernel parameters.

   ```
   cgroup_enable=memory cgroup_memory=1 systemd.unified_cgroup_hierarchy=0
   ```

   For information about viewing and setting kernel parameters for your device, see the documentation for your operating system and boot loader. Follow the instructions to permanently set the kernel parameters.

5. Install all other required dependencies on your device as indicated by the list of requirements in Device requirements.

Set up a Windows device

> **Note**
>
> This feature is available for v2.5.0 and later of the Greengrass nucleus component.

To set up a Windows device for AWS IoT Greengrass V2

1. Install the Java runtime, which AWS IoT Greengrass Core software requires to run. We recommend that you use Amazon Corretto or OpenJDK long-term support versions. Version 8 or higher is required.
2. Check whether Java is available on the PATH system variable, and add it if not. The LocalSystem account runs the AWS IoT Greengrass Core software, so you must add Java to the PATH system variable instead of the PATH user variable for your user. Do the following:
   a. Press the Windows key to open the start menu.
   b. Type **`environment variables`** to search for the system options from the start menu.
   c. In the start menu search results, choose **Edit the system environment variables** to open the **System properties** window.
   d. Choose **Environment variables...** to open the **Environment Variables** window.
   e. Under **System variables**, select **Path**, and then choose **Edit**. In the **Edit environment variable** window, you can view each path on a separate line.
   f. Check if the path to the Java installation's `bin` folder is present. The path might look similar to the following example.

      ```
      C:\\Program Files\\Amazon Corretto\\jdk11.0.13_8\\bin
      ```

   g. If the Java installation's `bin` folder is missing from **Path**, choose **New** to add it, and then choose **OK**.
3. Open the Windows Command Prompt (`cmd.exe`) as an administrator.
4. Create the default user in the LocalSystem account on the Windows device. Replace *`password`* with a secure password.

```
net user /add ggc_user password
```

Tip

Depending on your Windows configuration, the user's password might be set to expire at a date in the future. To ensure your Greengrass applications continue to operate, track when the password expires, and update it before it expires. You can also set the user's password to never expire.

- To check when a user and its password expire, run the following command.

  ```
  net user ggc_user | findstr /C:expires
  ```

- To set a user's password to never expire, run the following command.

  ```
  wmic UserAccount where "Name='ggc_user'" set PasswordExpires=False
  ```

- If you're using Windows 10 or later where the wmic command is deprecated, run the following PowerShell command.

  ```
  Get-CimInstance -Query "SELECT * from Win32_UserAccount WHERE name =
   'ggc_user'" | Set-CimInstance -Property @{PasswordExpires="False"}
  ```

5. Download and install the PsExec utility from Microsoft on the device.
6. Use the PsExec utility to store the user name and password for the default user in the Credential Manager instance for the LocalSystem account. Replace *password* with the user's password that you set earlier.

   ```
   psexec -s cmd /c cmdkey /generic:ggc_user /user:ggc_user /pass:password
   ```

   If the **PsExec License Agreement** opens, choose **Accept** to agree to the license and run the command.

Note

On Windows devices, the LocalSystem account runs the Greengrass nucleus, and you must use the PsExec utility to store the default user information in the LocalSystem

account. Using the Credential Manager application stores this information in the Windows account of the currently logged on user, instead of the LocalSystem account.

Download the AWS IoT Greengrass Core software

You can download the latest version of the AWS IoT Greengrass Core software from the following location:

- https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip

Note

You can download a specific version of the AWS IoT Greengrass Core software from the following location. Replace *version* with the version to download.

```
https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-version.zip
```

To download the AWS IoT Greengrass Core software

1. On your core device, download the AWS IoT Greengrass Core software to a file named `greengrass-nucleus-latest.zip`.

   Linux or Unix

   ```
   curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip > greengrass-nucleus-latest.zip
   ```

   Windows Command Prompt (CMD)

   ```
   curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip > greengrass-nucleus-latest.zip
   ```

   PowerShell

   ```
   iwr -Uri https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip -OutFile greengrass-nucleus-latest.zip
   ```

By downloading this software, you agree to the Greengrass Core Software License Agreement.

2. (Optional) To verify the Greengrass nucleus software signature

> **Note**
>
> This feature is available with Greengrass nucleus version 2.9.5 and later.

a. Use the following command to verify your Greengrass nucleus artifact's signature:

Linux or Unix

```
jarsigner -verify -certs -verbose greengrass-nucleus-latest.zip
```

Windows Command Prompt (CMD)

The file name might look different depending on the JDK version you install. Replace *jdk17.0.6_10* with the JDK version you installed.

```
"C:\\Program Files\\Amazon Corretto\\jdk17.0.6_10\\bin\\jarsigner.exe" -
verify -certs -verbose greengrass-nucleus-latest.zip
```

PowerShell

The file name might look different depending on the JDK version you install. Replace *jdk17.0.6_10* with the JDK version you installed.

```
'C:\\Program Files\\Amazon Corretto\\jdk17.0.6_10\\bin\\jarsigner.exe' -
verify -certs -verbose greengrass-nucleus-latest.zip
```

b. The `jarsigner` invocation yields output that indicates the results of the verification.

i. If the Greengrass nucleus zip file is signed, the output contains the following statement:

```
jar verified.
```

ii. If the Greengrass nucleus zip file isn't signed, the output contains the following statement:

```
jar is unsigned.
```

c. If you provided the Jarsigner `-certs` option along with `-verify` and `-verbose` options, the output also includes detailed signer certificate information.

3. Unzip the AWS IoT Greengrass Core software to a folder on your device. Replace *GreengrassInstaller* with the folder that you want to use.

   Linux or Unix

   ```
   unzip greengrass-nucleus-latest.zip -d GreengrassInstaller && rm greengrass-nucleus-latest.zip
   ```

   Windows Command Prompt (CMD)

   ```
   mkdir GreengrassInstaller && tar -xf greengrass-nucleus-latest.zip -C GreengrassInstaller && del greengrass-nucleus-latest.zip
   ```

   PowerShell

   ```
   Expand-Archive -Path greengrass-nucleus-latest.zip -DestinationPath .\GreengrassInstaller
   rm greengrass-nucleus-latest.zip
   ```

4. (Optional) Run the following command to see the version of the AWS IoT Greengrass Core software.

   ```
   java -jar ./GreengrassInstaller/lib/Greengrass.jar --version
   ```

Important

If you install a version of the Greengrass nucleus earlier than v2.4.0, don't remove this folder after you install the AWS IoT Greengrass Core software. The AWS IoT Greengrass Core software uses the files in this folder to run.

If you downloaded the latest version of the software, you install v2.4.0 or later, and you can remove this folder after you install the AWS IoT Greengrass Core software.

Download the AWS IoT fleet provisioning plugin

You can download the latest version of the AWS IoT fleet provisioning plugin from the following location:

- https://d2s8p88vqu9w66.cloudfront.net/releases/aws-greengrass-FleetProvisioningByClaim/fleetprovisioningbyclaim-latest.jar

Note

You can download a specific version of the AWS IoT fleet provisioning plugin from the following location. Replace *version* with the version to download. For more information about each version of the fleet provisioning plugin, see AWS IoT fleet provisioning plugin changelog.

```
https://d2s8p88vqu9w66.cloudfront.net/releases/aws-greengrass-
FleetProvisioningByClaim/fleetprovisioningbyclaim-version.jar
```

The fleet provisioning plugin is open source. To view its source code, see the AWS IoT fleet provisioning plugin on GitHub.

To download the AWS IoT fleet provisioning plugin

- On your device, download the AWS IoT fleet provisioning plugin to a file named `aws.greengrass.FleetProvisioningByClaim.jar`. Replace *GreengrassInstaller* with the folder that you want to use.

  Linux or Unix

  ```
  curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/aws-greengrass-
  FleetProvisioningByClaim/fleetprovisioningbyclaim-latest.jar
   > GreengrassInstaller/aws.greengrass.FleetProvisioningByClaim.jar
  ```

Windows Command Prompt (CMD)

```
curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/aws-greengrass-
FleetProvisioningByClaim/fleetprovisioningbyclaim-latest.jar
 > GreengrassInstaller/aws.greengrass.FleetProvisioningByClaim.jar
```

PowerShell

```
iwr -Uri https://d2s8p88vqu9w66.cloudfront.net/releases/aws-greengrass-
FleetProvisioningByClaim/fleetprovisioningbyclaim-latest.jar -
OutFile GreengrassInstaller/aws.greengrass.FleetProvisioningByClaim.jar
```

By downloading this software, you agree to the Greengrass Core Software License Agreement.

Install the AWS IoT Greengrass Core software

Run the installer with arguments that specify the following actions:

- Install from a partial configuration file that specifies to use the fleet provisioning plugin to provision AWS resources. The AWS IoT Greengrass Core software uses a configuration file that specifies the configuration of every Greengrass component on the device. The installer creates a complete configuration file from the partial configuration file that you provide and the AWS resources that the fleet provisioning plugin creates.
- Specify to use the `ggc_user` system user to run software components on the core device. On Linux devices, this command also specifies to use the `ggc_group` system group, and the installer creates the system user and group for you.
- Set up the AWS IoT Greengrass Core software as a system service that runs at boot. On Linux devices, this requires the Systemd init system.

> **Important**
>
> On Windows core devices, you must set up the AWS IoT Greengrass Core software as a system service.

For more information about the arguments that you can specify, see Installer arguments.

> **Note**
>
> If you are running AWS IoT Greengrass on a device with limited memory, you can control the amount of memory that AWS IoT Greengrass Core software uses. To control memory allocation, you can set JVM heap size options in the `jvmOptions` configuration parameter in your nucleus component. For more information, see Control memory allocation with JVM options.

To install the AWS IoT Greengrass Core software

1. Check the version of the AWS IoT Greengrass Core software.

   - Replace *GreengrassInstaller* with the path to the folder that contains the software.

   ```
   java -jar ./GreengrassInstaller/lib/Greengrass.jar --version
   ```

2. Use a text editor to create a configuration file named `config.yaml` to provide to the installer.

   For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

   ```
   nano GreengrassInstaller/config.yaml
   ```

   Copy the following YAML content into the file. This partial configuration file specifies parameters for the fleet provisioning plugin. For more information about the options that you can specify, see Configure the AWS IoT fleet provisioning plugin.

   Linux or Unix

   ```
   ---
   services:
     aws.greengrass.Nucleus:
       version: "2.18.4"
       configuration:
         fipsMode: "true"
         greengrassDataPlaneEndpoint: "iotData"
     aws.greengrass.FleetProvisioningByClaim:
   ```

```
    configuration:
      rootPath: "/greengrass/v2"
      awsRegion: "us-west-2"
      iotDataEndpoint: "data.iot-fips.us-west-2.amazonaws.com"
      iotCredEndpoint: "data.credentials.iot-fips.us-west-2.amazonaws.com"
      iotRoleAlias: "GreengrassCoreTokenExchangeRoleAlias"
      provisioningTemplate: "GreengrassFleetProvisioningTemplate"
      claimCertificatePath: "/greengrass/v2/claim-certs/claim.pem.crt"
      claimCertificatePrivateKeyPath: "/greengrass/v2/claim-certs/
claim.private.pem.key"
      rootCaPath: "/greengrass/v2/AmazonRootCA1.pem"
      templateParameters:
        ThingName: "MyGreengrassCore"
        ThingGroupName: "MyGreengrassCoreGroup"
```

Windows

```
---
services:
  aws.greengrass.Nucleus:
    version: "2.18.4"
  aws.greengrass.FleetProvisioningByClaim:
    configuration:
      rootPath: "C:\\greengrass\\v2"
      awsRegion: "us-west-2"
      iotDataEndpoint: "device-data-prefix-ats.iot.us-west-2.amazonaws.com"
      iotCredentialEndpoint: "device-credentials-prefix.credentials.iot.us-
west-2.amazonaws.com"
      iotRoleAlias: "GreengrassCoreTokenExchangeRoleAlias"
      provisioningTemplate: "GreengrassFleetProvisioningTemplate"
      claimCertificatePath: "C:\\greengrass\\v2\\claim-certs\\claim.pem.crt"
      claimCertificatePrivateKeyPath: "C:\\greengrass\\v2\\claim-certs\
\claim.private.pem.key"
      rootCaPath: "C:\\greengrass\\v2\\AmazonRootCA1.pem"
      templateParameters:
        ThingName: "MyGreengrassCore"
        ThingGroupName: "MyGreengrassCoreGroup"
```

Then, do the following:

- Replace *2.18.4* with the version of the AWS IoT Greengrass Core software.

- Replace each instance of */greengrass/v2* or *C:\greengrass\v2* with the Greengrass root folder.

  > **Note**
  >
  > On Windows devices, you must specify path separators as double backslashes (\\), such as `C:\\greengrass\\v2`.

- Replace *us-west-2* with the AWS Region where you created the provisioning template and other resources.
- Replace the `iotDataEndpoint` with your AWS IoT data endpoint.
- Replace the `iotCredentialEndpoint` with your AWS IoT credentials endpoint.
- Replace *GreengrassCoreTokenExchangeRoleAlias* with the name of the token exchange role alias.
- Replace *GreengrassFleetProvisioningTemplate* with the name of the fleet provisioning template.
- Replace the `claimCertificatePath` with the path to the claim certificate on the device.
- Replace the `claimCertificatePrivateKeyPath` with the path to the claim certificate private key on the device.
- Replace the template parameters (`templateParameters`) with the values to use to provision the device. This example refers to the example template that defines `ThingName` and `ThingGroupName` parameters.

3. Run the installer. Specify `--trusted-plugin` to provide the fleet provisioning plugin, and specify `--init-config` to provide the configuration file.

   - Replace */greengrass/v2* with the Greengrass root folder.
   - Replace each instance of *GreengrassInstaller* with the folder where you unpacked the installer.

   Linux or Unix

   ```
   sudo -E java -Droot="/greengrass/v2" -Dlog.store=FILE \
     -jar ./GreengrassInstaller/lib/Greengrass.jar \
     --trusted-plugin ./GreengrassInstaller/
   aws.greengrass.FleetProvisioningByClaim.jar \
     --init-config ./GreengrassInstaller/config.yaml \
   ```

```
  --component-default-user ggc_user:ggc_group \
  --setup-system-service true
```

Windows Command Prompt (CMD)

```
java -Droot="C:\greengrass\v2" "-Dlog.store=FILE" ^
  -jar ./GreengrassInstaller/lib/Greengrass.jar ^
  --trusted-plugin ./GreengrassInstaller/
aws.greengrass.FleetProvisioningByClaim.jar ^
  --init-config ./GreengrassInstaller/config.yaml ^
  --component-default-user ggc_user ^
  --setup-system-service true
```

PowerShell

```
java -Droot="C:\greengrass\v2" "-Dlog.store=FILE" `
  -jar ./GreengrassInstaller/lib/Greengrass.jar `
  --trusted-plugin ./GreengrassInstaller/
aws.greengrass.FleetProvisioningByClaim.jar `
  --init-config ./GreengrassInstaller/config.yaml `
  --component-default-user ggc_user `
  --setup-system-service true
```

Important

On Windows core devices, you must specify `--setup-system-service true` to set up the AWS IoT Greengrass Core software as a system service.

If you specify `--setup-system-service true`, the installer prints `Successfully set up Nucleus as a system service` if it set up and ran the software as a system service. Otherwise, the installer doesn't output any message if it installs the software successfully.

Note

You can't use the `deploy-dev-tools` argument to deploy local development tools when you run the installer without the `--provision true` argument.

For information about deploying the Greengrass CLI directly on your device, see Greengrass Command Line Interface.

4. Verify the installation by viewing the files in the root folder.

   Linux or Unix

   ```
   ls /greengrass/v2
   ```

   Windows Command Prompt (CMD)

   ```
   dir C:\greengrass\v2
   ```

   PowerShell

   ```
   ls C:\greengrass\v2
   ```

   If the installation succeeded, the root folder contains several folders, such as `config`, `packages`, and `logs`.

If you installed the AWS IoT Greengrass Core software as a system service, the installer runs the software for you. Otherwise, you must run the software manually. For more information, see Run the AWS IoT Greengrass Core software.

For more information about how to configure and use the software and AWS IoT Greengrass, see the following:

- Configure the AWS IoT Greengrass Core software
- Develop AWS IoT Greengrass components
- Deploy AWS IoT Greengrass components to devices
- Greengrass Command Line Interface

Install FIPS endpoints with auto resource provisioning

The AWS IoT Greengrass Core software includes an installer that sets up your device as a Greengrass core device. To set up a device quickly, the installer can provision the AWS IoT thing,

AWS IoT thing group, IAM role, and AWS IoT role alias that the core device requires to operate. The installer can also deploy the local development tools to the core device, so you can use the device to develop and test custom software components. The installer requires AWS credentials to provision these resources and create the deployment.

If you can't provide AWS credentials to the device, you can provision the AWS resources that the core device requires to operate. You can also deploy the development tools to a core device to use as a development device. This enables you to provide fewer permissions to the device when you run the installer. For more information, see Install AWS IoT Greengrass Core software with manual resource provisioning.

> **Important**
>
> Before you download the AWS IoT Greengrass Core software, check that your core device meets the requirements to install and run the AWS IoT Greengrass Core software v2.0.

Topics

- Set up the device environment
- Provide AWS credentials to the device
- Download the AWS IoT Greengrass Core software
- Install the AWS IoT Greengrass Core software

Set up the device environment

Follow the steps in this section to set up a Linux or Windows device to use as your AWS IoT Greengrass core device.

Set up a Linux device

To set up a Linux device for AWS IoT Greengrass V2

1. Install the Java runtime, which AWS IoT Greengrass Core software requires to run. We recommend that you use Amazon Corretto or OpenJDK long-term support versions. Version 8 or higher is required. The following commands show you how to install OpenJDK on your device.

   - For Debian-based or Ubuntu-based distributions:

```
sudo apt install default-jdk
```

- For Red Hat-based distributions:

```
sudo yum install java-11-openjdk-devel
```

- For Amazon Linux 2:

```
sudo amazon-linux-extras install java-openjdk11
```

- For Amazon Linux 2023:

```
sudo dnf install java-11-amazon-corretto -y
```

When the installation completes, run the following command to verify that Java runs on your Linux device.

```
java -version
```

The command prints the version of Java that runs on the device. For example, on a Debian-based distribution, the output might look similar to the following sample.

```
openjdk version "11.0.9.1" 2020-11-04
OpenJDK Runtime Environment (build 11.0.9.1+1-post-Debian-1deb10u2)
OpenJDK 64-Bit Server VM (build 11.0.9.1+1-post-Debian-1deb10u2, mixed mode)
```

2. (Optional) Create the default system user and group that runs components on the device. You can also choose to let the AWS IoT Greengrass Core software installer create this user and group during installation with the `--component-default-user` installer argument. For more information, see Installer arguments.

```
sudo useradd --system --create-home ggc_user
sudo groupadd --system ggc_group
```

3. Verify that the user that runs the AWS IoT Greengrass Core software (typically `root`), has permission to run `sudo` with any user and any group.

   a. Run the following command to open the `/etc/sudoers` file.

```
sudo visudo
```

b. Verify that the permission for the user looks like the following example.

```
root    ALL=(ALL:ALL) ALL
```

4. (Optional) To run containerized Lambda functions, you must enable cgroups v1, and you must enable and mount the *memory* and *devices* cgroups. If you don't plan to run containerized Lambda functions, you can skip this step.

   To enable these cgroups options, boot the device with the following Linux kernel parameters.

   ```
   cgroup_enable=memory cgroup_memory=1 systemd.unified_cgroup_hierarchy=0
   ```

   For information about viewing and setting kernel parameters for your device, see the documentation for your operating system and boot loader. Follow the instructions to permanently set the kernel parameters.

5. Install all other required dependencies on your device as indicated by the list of requirements in Device requirements.

Set up a Windows device

> **Note**
>
> This feature is available for v2.5.0 and later of the Greengrass nucleus component.

To set up a Windows device for AWS IoT Greengrass V2

1. Install the Java runtime, which AWS IoT Greengrass Core software requires to run. We recommend that you use Amazon Corretto or OpenJDK long-term support versions. Version 8 or higher is required.
2. Check whether Java is available on the PATH system variable, and add it if not. The LocalSystem account runs the AWS IoT Greengrass Core software, so you must add Java to the PATH system variable instead of the PATH user variable for your user. Do the following:

   a. Press the Windows key to open the start menu.

b. Type **`environment variables`** to search for the system options from the start menu.

c. In the start menu search results, choose **Edit the system environment variables** to open the **System properties** window.

d. Choose **Environment variables...** to open the **Environment Variables** window.

e. Under **System variables**, select **Path**, and then choose **Edit**. In the **Edit environment variable** window, you can view each path on a separate line.

f. Check if the path to the Java installation's `bin` folder is present. The path might look similar to the following example.

```
C:\\Program Files\\Amazon Corretto\\jdk11.0.13_8\\bin
```

g. If the Java installation's `bin` folder is missing from **Path**, choose **New** to add it, and then choose **OK**.

3. Open the Windows Command Prompt (`cmd.exe`) as an administrator.

4. Create the default user in the LocalSystem account on the Windows device. Replace *`password`* with a secure password.

```
net user /add ggc_user password
```

Tip

Depending on your Windows configuration, the user's password might be set to expire at a date in the future. To ensure your Greengrass applications continue to operate, track when the password expires, and update it before it expires. You can also set the user's password to never expire.

- To check when a user and its password expire, run the following command.

```
net user ggc_user | findstr /C:expires
```

- To set a user's password to never expire, run the following command.

```
wmic UserAccount where "Name='ggc_user'" set PasswordExpires=False
```

- If you're using Windows 10 or later where the wmic command is deprecated, run the following PowerShell command.

```
Get-CimInstance -Query "SELECT * from Win32_UserAccount WHERE name =
 'ggc_user'" | Set-CimInstance -Property @{PasswordExpires="False"}
```

5. Download and install the PsExec utility from Microsoft on the device.
6. Use the PsExec utility to store the user name and password for the default user in the Credential Manager instance for the LocalSystem account. Replace *password* with the user's password that you set earlier.

   ```
   psexec -s cmd /c cmdkey /generic:ggc_user /user:ggc_user /pass:password
   ```

   If the **PsExec License Agreement** opens, choose **Accept** to agree to the license and run the command.

   > **Note**
   >
   > On Windows devices, the LocalSystem account runs the Greengrass nucleus, and you must use the PsExec utility to store the default user information in the LocalSystem account. Using the Credential Manager application stores this information in the Windows account of the currently logged on user, instead of the LocalSystem account.

Provide AWS credentials to the device

Provide your AWS credentials to your device so that the installer can provision the required AWS resources. For more information about the required permissions, see Minimal IAM policy for installer to provision resources.

To provide AWS credentials to the device

- Provide your AWS credentials to the device so that the installer can provision the AWS IoT and IAM resources for your core device. To increase security, we recommend that you get temporary credentials for an IAM role that allows only the minimum permissions necessary to provision. For more information, see Minimal IAM policy for installer to provision resources.

> **Note**
>
> The installer doesn't save or store your credentials.

On your device, do one of the following to retrieve credentials and make them available to the AWS IoT Greengrass Core software installer:

- (Recommended) Use temporary credentials from AWS IAM Identity Center

  a. Provide the access key ID, secret access key, and session token from the IAM Identity Center. For more information, see **Manual credential refresh** in Getting and refreshing temporary credentials in the *IAM Identity Center user guide*.

  b. Run the following commands to provide the credentials to the AWS IoT Greengrass Core software.

  Linux or Unix

  ```
  export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
  export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
  export AWS_SESSION_TOKEN=AQoDYXdzEJr1K...o5OytwEXAMPLE=
  ```

  Windows Command Prompt (CMD)

  ```
  set AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
  set AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
  set AWS_SESSION_TOKEN=AQoDYXdzEJr1K...o5OytwEXAMPLE=
  ```

  PowerShell

  ```
  $env:AWS_ACCESS_KEY_ID="AKIAIOSFODNN7EXAMPLE"
  $env:AWS_SECRET_ACCESS_KEY="wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"
  $env:AWS_SESSION_TOKEN="AQoDYXdzEJr1K...o5OytwEXAMPLE="
  ```

- Use temporary security credentials from an IAM role:

  a. Provide the access key ID, secret access key, and session token from an IAM role that you assume. For more information about how to retrieve these credentials, see Requesting temporary security credentials in the *IAM User Guide*.

b. Run the following commands to provide the credentials to the AWS IoT Greengrass Core software.

Linux or Unix

```
export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
export AWS_SESSION_TOKEN=AQoDYXdzEJr1K...o5OytwEXAMPLE=
```

Windows Command Prompt (CMD)

```
set AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
set AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
set AWS_SESSION_TOKEN=AQoDYXdzEJr1K...o5OytwEXAMPLE=
```

PowerShell

```
$env:AWS_ACCESS_KEY_ID="AKIAIOSFODNN7EXAMPLE"
$env:AWS_SECRET_ACCESS_KEY="wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"
$env:AWS_SESSION_TOKEN="AQoDYXdzEJr1K...o5OytwEXAMPLE="
```

- Use long-term credentials from an IAM user:

a. Provide the access key ID and secret access key for your IAM user. You can create an IAM user for provisioning that you later delete. For the IAM policy to give the user, see Minimal IAM policy for installer to provision resources. For more information about how to retrieve long-term credentials, see Managing access keys for IAM users in the *IAM User Guide*.

b. Run the following commands to provide the credentials to the AWS IoT Greengrass Core software.

Linux or Unix

```
export AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
export AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

Windows Command Prompt (CMD)

```
set AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
```

```
set AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

PowerShell

```
$env:AWS_ACCESS_KEY_ID="AKIAIOSFODNN7EXAMPLE"
$env:AWS_SECRET_ACCESS_KEY="wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"
```

c. (Optional) If you created an IAM user to provision your Greengrass device, delete the user.

d. (Optional) If you used the access key ID and secret access key from an existing IAM user, update the keys for the user so that they are no longer valid. For more information, see Updating access keys in the *AWS Identity and Access Management user guide*.

Download the AWS IoT Greengrass Core software

You can download the latest version of the AWS IoT Greengrass Core software from the following location:

- https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-latest.zip

> **Note**
>
> You can download a specific version of the AWS IoT Greengrass Core software from the following location. Replace *version* with the version to download.
>
> ```
> https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-version.zip
> ```

To download the AWS IoT Greengrass Core software

1. On your core device, download the AWS IoT Greengrass Core software to a file named `greengrass-nucleus-latest.zip`.

Linux or Unix

```
curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-
latest.zip > greengrass-nucleus-latest.zip
```

Windows Command Prompt (CMD)

```
curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-
latest.zip > greengrass-nucleus-latest.zip
```

PowerShell

```
iwr -Uri https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-nucleus-
latest.zip -OutFile greengrass-nucleus-latest.zip
```

By downloading this software, you agree to the Greengrass Core Software License Agreement.

2. (Optional) To verify the Greengrass nucleus software signature

> **Note**
>
> This feature is available with Greengrass nucleus version 2.9.5 and later.

a. Use the following command to verify your Greengrass nucleus artifact's signature:

Linux or Unix

```
jarsigner -verify -certs -verbose greengrass-nucleus-latest.zip
```

Windows Command Prompt (CMD)

The file name might look different depending on the JDK version you install. Replace *jdk17.0.6_10* with the JDK version you installed.

```
"C:\\Program Files\\Amazon Corretto\\jdk17.0.6_10\\bin\\jarsigner.exe" -
verify -certs -verbose greengrass-nucleus-latest.zip
```

PowerShell

The file name might look different depending on the JDK version you install. Replace *jdk17.0.6_10* with the JDK version you installed.

```
'C:\\Program Files\\Amazon Corretto\\jdk17.0.6_10\\bin\\jarsigner.exe' -
verify -certs -verbose greengrass-nucleus-latest.zip
```

b. The `jarsigner` invocation yields output that indicates the results of the verification.

   i. If the Greengrass nucleus zip file is signed, the output contains the following statement:

   ```
   jar verified.
   ```

   ii. If the Greengrass nucleus zip file isn't signed, the output contains the following statement:

   ```
   jar is unsigned.
   ```

c. If you provided the Jarsigner `-certs` option along with `-verify` and `-verbose` options, the output also includes detailed signer certificate information.

3. Unzip the AWS IoT Greengrass Core software to a folder on your device. Replace *GreengrassInstaller* with the folder that you want to use.

   Linux or Unix

   ```
   unzip greengrass-nucleus-latest.zip -d GreengrassInstaller && rm greengrass-
   nucleus-latest.zip
   ```

   Windows Command Prompt (CMD)

   ```
   mkdir GreengrassInstaller && tar -xf greengrass-nucleus-latest.zip -
   C GreengrassInstaller && del greengrass-nucleus-latest.zip
   ```

   PowerShell

   ```
   Expand-Archive -Path greengrass-nucleus-latest.zip -DestinationPath .\
   \GreengrassInstaller
   ```

```
rm greengrass-nucleus-latest.zip
```

4. (Optional) Run the following command to see the version of the AWS IoT Greengrass Core software.

```
java -jar ./GreengrassInstaller/lib/Greengrass.jar --version
```

Important

If you install a version of the Greengrass nucleus earlier than v2.4.0, don't remove this folder after you install the AWS IoT Greengrass Core software. The AWS IoT Greengrass Core software uses the files in this folder to run.
If you downloaded the latest version of the software, you install v2.4.0 or later, and you can remove this folder after you install the AWS IoT Greengrass Core software.

Install the AWS IoT Greengrass Core software

Run the installer with arguments that specify to do the following:

- Create the AWS resources that the core device requires to operate.
- Specify to use the `ggc_user` system user to run software components on the core device. On Linux devices, this command also specifies to use the `ggc_group` system group, and the installer creates the system user and group for you.
- Set up the AWS IoT Greengrass Core software as a system service that runs at boot. On Linux devices, this requires the Systemd init system.

Important

On Windows core devices, you must set up the AWS IoT Greengrass Core software as a system service.

To set up a development device with local development tools, specify the `--deploy-dev-tools true` argument. The local development tools can take up to a minute to deploy after the installation completes.

For more information about the arguments that you can specify, see Installer arguments.

> **Note**
>
> If you are running AWS IoT Greengrass on a device with limited memory, you can control the amount of memory that AWS IoT Greengrass Core software uses. To control memory allocation, you can set JVM heap size options in the `jvmOptions` configuration parameter in your nucleus component. For more information, see Control memory allocation with JVM options.

To install the AWS IoT Greengrass Core software

1. Use a text editor to create a configuration file named `config.yaml` to provide to the installer.

   For example, on a Linux-based system, you can run the following command to use GNU nano to create the file.

   ```
   nano GreengrassInstaller/config.yaml
   ```

   Copy the following YAML content into the file. This partial configuration file specifies system parameters and Greengrass nucleus parameters.

   ```
   ---
   services:
     aws.greengrass.Nucleus:
       configuration:
         fipsMode: "true"
         iotDataEndpoint: "data.iot-fips.us-west-2.amazonaws.com"
         iotCredEndpoint: "data.credentials.iot-fips.us-west-2.amazonaws.com"
         greengrassDataPlaneEndpoint: "iotData"
   ```

   - Replace *us-west-2* with the AWS Region where you created the resources.
   - Replace the *iotDataEndpoint* with your AWS IoT data endpoint.
   - Replace the *iotCredEndpoint* with your AWS IoT credentials endpoint.
2. Run the AWS IoT Greengrass Core installer. Replace argument values in your command as follows.

> **Note**
>
> Windows has a path length limitation of 260 characters. If you are using Windows, use a root folder like `C:\greengrass\v2` or `D:\greengrass\v2` to keep the Greengrass components paths under the 260 character limit.

a. */greengrass/v2* or *C:\greengrass\v2*: The path to the root folder to use to install the AWS IoT Greengrass Core software.

b. *GreengrassInstaller*. The path to the folder where you unpacked the AWS IoT Greengrass Core software installer.

c. *region*. The AWS Region in which to find or create resources.

d. *MyGreengrassCore*. The name of the AWS IoT thing for your Greengrass core device. If the thing doesn't exist, the installer creates it. The installer downloads the certificates to authenticate as the AWS IoT thing. For more information, see Device authentication and authorization for AWS IoT Greengrass.

> **Note**
>
> The thing name can't contain colon (`:`) characters.

e. *MyGreengrassCoreGroup*. The name of AWS IoT thing group for your Greengrass core device. If the thing group doesn't exist, the installer creates it and adds the thing to it. If the thing group exists and has an active deployment, the core device downloads and runs the software that the deployment specifies.

> **Note**
>
> The thing group name can't contain colon (`:`) characters.

f. *GreengrassV2IoTThingPolicy*. The name of the AWS IoT policy that allows the Greengrass core devices to communicate with AWS IoT and AWS IoT Greengrass. If the AWS IoT policy doesn't exist, the installer creates a permissive AWS IoT policy with this name. You can restrict this policy's permissions for you use case. For more information, see Minimal AWS IoT policy for AWS IoT Greengrass V2 core devices.

g. *GreengrassV2TokenExchangeRole*. The name of the IAM role that allows the Greengrass core device to get temporary AWS credentials. If the role doesn't exist, the installer creates it and creates and attaches a policy named *GreengrassV2TokenExchangeRole*Access. For more information, see Authorize core devices to interact with AWS services.

h. *GreengrassCoreTokenExchangeRoleAlias*. The alias to the IAM role that allows the Greengrass core device to get temporary credentials later. If the role alias doesn't exist, the installer creates it and points it to the IAM role that you specify. For more information, see Authorize core devices to interact with AWS services.

Linux or Unix

```
sudo -E java -Droot="/greengrass/v2" -Dlog.store=FILE \
  -jar ./GreengrassInstaller/lib/Greengrass.jar \
  --aws-region region \
  --thing-name MyGreengrassCore \
  --thing-group-name MyGreengrassCoreGroup \
  --thing-policy-name GreengrassV2IoTThingPolicy \
  --tes-role-name GreengrassV2TokenExchangeRole \
  --tes-role-alias-name GreengrassCoreTokenExchangeRoleAlias \
  --component-default-user ggc_user:ggc_group \
  --provision true \
  --init-config ./GreengrassInstaller/config.yaml \
  --setup-system-service true
```

Windows Command Prompt (CMD)

```
java -Droot="C:\greengrass\v2" "-Dlog.store=FILE" ^
  -jar ./GreengrassInstaller/lib/Greengrass.jar ^
  --aws-region region ^
  --thing-name MyGreengrassCore ^
  --thing-group-name MyGreengrassCoreGroup ^
  --thing-policy-name GreengrassV2IoTThingPolicy ^
  --tes-role-name GreengrassV2TokenExchangeRole ^
  --tes-role-alias-name GreengrassCoreTokenExchangeRoleAlias ^
  --component-default-user ggc_user ^
  --provision true ^
  --setup-system-service true
```

PowerShell

```
java -Droot="C:\greengrass\v2" "-Dlog.store=FILE" `
  -jar ./GreengrassInstaller/lib/Greengrass.jar `
  --aws-region region `
  --thing-name MyGreengrassCore `
  --thing-group-name MyGreengrassCoreGroup `
  --thing-policy-name GreengrassV2IoTThingPolicy `
  --tes-role-name GreengrassV2TokenExchangeRole `
  --tes-role-alias-name GreengrassCoreTokenExchangeRoleAlias `
  --component-default-user ggc_user `
  --provision true `
  --setup-system-service true
```

Important

On Windows core devices, you must specify `--setup-system-service true` to set up the AWS IoT Greengrass Core software as a system service.

The installer prints the following messages if it succeeds:

- If you specify `--provision`, the installer prints `Successfully configured Nucleus with provisioned resource details` if it configured the resources successfully.
- If you specify `--deploy-dev-tools`, the installer prints `Configured Nucleus to deploy aws.greengrass.Cli component` if it created the deployment successfully.
- If you specify `--setup-system-service true`, the installer prints `Successfully set up Nucleus as a system service` if it set up and ran the software as a service.
- If you don't specify `--setup-system-service true`, the installer prints `Launched Nucleus successfully` if it succeeded and ran the software.

3. Skip this step if you installed Greengrass nucleus v2.0.4 or later. If you downloaded the latest version of the software, you installed v2.0.4 or later.

   Run the following command to set the required file permissions for your AWS IoT Greengrass Core software root folder. Replace */greengrass/v2* with the root folder that you specified in your installation command, and replace */greengrass* with the parent folder for your root folder.

```
sudo chmod 755 /greengrass/v2 && sudo chmod 755 /greengrass
```

If you installed the AWS IoT Greengrass Core software as a system service, the installer runs the software for you. Otherwise, you must run the software manually. For more information, see Run the AWS IoT Greengrass Core software.

Note

By default, the IAM role that the installer creates doesn't allow access to component artifacts in S3 buckets. To deploy custom components that define artifacts in Amazon S3, you must add permissions to the role to allow your core device to retrieve component artifacts. For more information, see Allow access to S3 buckets for component artifacts. If you don't yet have an S3 bucket for component artifacts, you can add these permissions later after you create a bucket.

For more information about how to configure and use the software and AWS IoT Greengrass, see the following:

- Configure the AWS IoT Greengrass Core software
- Develop AWS IoT Greengrass components
- Deploy AWS IoT Greengrass components to devices
- Greengrass Command Line Interface

FIPS compliance first party components

aws.greengrass.Nucleus	data.iot-fips. *us-east-1* .amazonaws.com
	greengrass-fips. *us-east-1* .amazonaws.com

	data.credentials.iot-fips. *us-east-1* .amazonaws.com
aws.greengrass.TokenExchangeService	data.credentials.iot-fips. *us-east-1* .amazonaws.com
aws.greengrass.Cli	
aws.greengrass.StreamManager	• kinesis-fips. *us-east-1* .amazonaws.com • data.iotsitewise-fips. *us-east-1* .amazonaws.com • s3-fips.*us-east-1* .amazonaws.com
aws.greengrass.LogManager	logs-fips.*us-east-1* .amazonaws.com
aws.greengrass.crypto.Pkcs11Provider	
aws.greengrass.ShadowManager	
aws.greengrass.DockerApplicationManager	ecr-fips.*us-east-1* .amazonaws.com
aws.greengrass.SecretManager	secretsmanager-fips.*us-east-1* .amazonaws.com
aws.greengrass.telemetry.NucleusEmitter	
aws.greengrass.clientdevices.IPDetector	
aws.greengrass.DiskSpooler	

Resilience in AWS IoT Greengrass

The AWS global infrastructure is built around Amazon Web Services Regions and Availability Zones. Each AWS Region provides multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information, see AWS Global Infrastructure.

In addition to the AWS global infrastructure, AWS IoT Greengrass offers several features to help support your data resiliency and backup needs.

- You can configure a Greengrass core device to write logs to the local file system and to CloudWatch Logs. If the core device loses connectivity, it can continue to log messages on the file system. When it reconnects, it writes the log messages to CloudWatch Logs. For more information, see Monitor AWS IoT Greengrass logs.
- If a core device loses power during a deployment, it resumes the deployment after the AWS IoT Greengrass Core software starts again.
- If a core device loses internet connectivity, Greengrass client devices can continue to communicate over the local network.
- You can author Greengrass components that read stream manager streams and send the data to local storage destinations.

Infrastructure security in AWS IoT Greengrass

As a managed service, AWS IoT Greengrass is protected by the AWS global network security procedures that are described in the Amazon Web Services: Overview of Security Processes whitepaper.

You use AWS published API calls to access AWS IoT Greengrass through the network. Clients must support Transport Layer Security (TLS) 1.2 or later. We recommend TLS 1.3 or later. Clients must also support cipher suites with perfect forward secrecy (PFS), such as Ephemeral Diffie-Hellman (DHE) or Elliptic Curve Ephemeral Diffie-Hellman (ECDHE). Most modern systems such as Java 7 and later support these modes.

Requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the AWS Security Token Service (AWS STS) to generate temporary security credentials to sign requests.

In an AWS IoT Greengrass environment, devices use X.509 certificates and cryptographic keys to connect and authenticate to the AWS Cloud. For more information, see the section called "Device authentication and authorization".

Configuration and vulnerability analysis in AWS IoT Greengrass

IoT environments can consist of large numbers of devices that have diverse capabilities, are long-lived, and are geographically distributed. These characteristics make device setup complex and error-prone. And because devices are often constrained in computational power, memory, and storage capabilities, this limits the use of encryption and other forms of security on the devices themselves. Also, devices often use software with known vulnerabilities. These factors make IoT devices an attractive target for hackers and make it difficult to secure them on an ongoing basis.

AWS IoT Device Defender addresses these challenges by providing tools to identify security issues and deviations from best practices. You can use AWS IoT Device Defender to analyze, audit, and monitor connected devices to detect abnormal behavior, and mitigate security risks. AWS IoT Device Defender can audit devices to ensure they adhere to security best practices and detect abnormal behavior on devices. This makes it possible to enforce consistent security policies across your devices and respond quickly when devices are compromised. IFor more information, see the following topics:

- The Device Defender component
- AWS IoT Device Defender in the *AWS IoT Core Developer Guide*.

In AWS IoT Greengrass environments, you should be aware of the following considerations:

- It's your reponsibility to secure your physical devices, the file system on your devices, and the local network.
- AWS IoT Greengrass doesn't enforce network isolation for user-defined Greengrass components, whether or not they run in a Greengrass container. Therefore, it's possible for Greengrass components to communicate with any other process running in the system or outside over network.

Code integrity in AWS IoT Greengrass V2

AWS IoT Greengrass deploys software components from the AWS Cloud to devices that run the AWS IoT Greengrass Core software. These software components include AWS-provided components and custom components that you upload to your AWS account. Every component is composed of a recipe. The recipe defines the component's metadata, and any number of artifacts, which are component binaries, such as compiled code and static resources. Component artifacts are stored in Amazon S3.

As you develop and deploy Greengrass components, you follow these basic steps that work with component artifacts in your AWS account and on your devices:

1. Create and upload artifacts to S3 buckets.
2. Create a component from a recipe and artifacts in the AWS IoT Greengrass service, which calculates a cryptographic hash of each artifact.
3. Deploy a component to Greengrass core devices, which download and verify the integrity of each artifact.

AWS is responsible for maintaining the integrity of artifacts after you upload artifacts to S3 buckets, including when you deploy components to Greengrass core devices. You are responsible for securing software artifacts before you upload the artifacts to S3 buckets. You are also responsible for securing access to resources in your AWS account, including the S3 buckets where you upload component artifacts.

> **Note**
>
> Amazon S3 provides a feature called S3 Object Lock that you can use to protect against changes to component artifacts in S3 buckets your AWS account. You can use S3 Object Lock to prevent component artifacts from being deleted or overwritten. For more information, see Using S3 Object Lock in the *Amazon Simple Storage Service User Guide*.

When AWS publishes a public component, and when you upload a custom component, AWS IoT Greengrass calculates a cryptographic digest for each component artifact. AWS IoT Greengrass updates the component recipe to include each artifact's digest and the hash algorithm used to calculate that digest. This digest guarantees the integrity of the artifact, because if the artifact changes in the AWS Cloud or during download, its file digest won't match the digest that AWS IoT

Greengrass stores in the component recipe. For more information, see Artifacts in the component recipe reference.

When you deploy a component to a core device, the AWS IoT Greengrass Core software downloads the component recipe and each component artifact that the recipe defines. The AWS IoT Greengrass Core software calculates the digest of each downloaded artifact file and compares it with that artifact's digest in the recipe. If the digests don't match, the deployment fails, and the AWS IoT Greengrass Core software deletes the downloaded artifacts from the device's file system. For more information about how connections between core devices and AWS IoT Greengrass are secured, see Encryption in transit.

You are responsible for securing component artifact files on your core devices' file systems. The AWS IoT Greengrass Core software saves artifacts to the `packages` folder in the Greengrass root folder. You can use AWS IoT Device Defender to analyze, audit, and monitor core devices. For more information, see Configuration and vulnerability analysis in AWS IoT Greengrass.

AWS IoT Greengrass and interface VPC endpoints (AWS PrivateLink)

You can establish a private connection between your VPC and the AWS IoT Greengrass control plane by creating an *interface VPC endpoint*. You can use this endpoint to manage components, deployments, and core devices in the AWS IoT Greengrass service. Interface endpoints are powered by AWS PrivateLink, a technology that enables you to access AWS IoT Greengrass APIs privately without an internet gateway, NAT device, VPN connection, or AWS Direct Connect connection. Instances in your VPC don't need public IP addresses to communicate with AWS IoT Greengrass APIs. Traffic between your VPC and AWS IoT Greengrass does not leave the Amazon network.

Each interface endpoint is represented by one or more Elastic Network Interfaces in your subnets.

For more information, see Interface VPC endpoints (AWS PrivateLink) in the *Amazon VPC User Guide*.

Topics

- Considerations for AWS IoT Greengrass VPC endpoints
- Create an interface VPC endpoint for AWS IoT Greengrass control plane operations
- Creating a VPC endpoint policy for AWS IoT Greengrass
- Operate an AWS IoT Greengrass core device in VPC

Considerations for AWS IoT Greengrass VPC endpoints

Before you set up an interface VPC endpoint for AWS IoT Greengrass, review Interface endpoint properties and limitations in the *Amazon VPC User Guide*. Additionally, be aware of the following considerations:

- AWS IoT Greengrass supports making calls to all of its control plane API actions from your VPC. The control plane includes operations such as CreateDeployment and ListEffectiveDeployments. The control plane does *not* include operations such as ResolveComponentCandidates and Discover, which are data plane operations.
- VPC endpoints for AWS IoT Greengrass are currently not supported in AWS China Regions.

Create an interface VPC endpoint for AWS IoT Greengrass control plane operations

You can create a VPC endpoint for the AWS IoT Greengrass control plane using either the Amazon VPC console or the AWS Command Line Interface (AWS CLI). For more information, see Creating an interface endpoint in the *Amazon VPC User Guide*.

Create a VPC endpoint for AWS IoT Greengrass using the following service name:

- com.amazonaws.*region*.greengrass

If you enable private DNS for the endpoint, you can make API requests to AWS IoT Greengrass using its default DNS name for the Region, for example, `greengrass.us-east-1.amazonaws.com`. Private DNS is enabled by default.

For more information, see Accessing a service through an interface endpoint in the *Amazon VPC User Guide*.

Creating a VPC endpoint policy for AWS IoT Greengrass

You can attach an endpoint policy to your VPC endpoint that controls access to AWS IoT Greengrass control plane operations. The policy specifies the following information:

- The principal that can perform actions.
- The actions that the principal can perform.

- The resources that the principal can perform actions on.

For more information, see Controlling access to services with VPC endpoints in the *Amazon VPC User Guide*.

Example: VPC endpoint policy for AWS IoT Greengrass actions

The following is an example of an endpoint policy for AWS IoT Greengrass. When attached to an endpoint, this policy grants access to the listed AWS IoT Greengrass actions for all principals on all resources.

```
{
    "Statement": [
        {
            "Principal": "*",
            "Effect": "Allow",
            "Action": [
                "greengrass:CreateDeployment",
                "greengrass:ListEffectiveDeployments"
            ],
            "Resource": "*"
        }
    ]
}
```

Operate an AWS IoT Greengrass core device in VPC

You can operate a Greengrass core device and perform deployments in VPC without public internet access. At a minimum, you must set up the following VPC endpoints with the corresponding DNS aliases. For more information about how to create and use VPC endpoints, see Create a VPC endpoint in the *Amazon VPC User Guide*.

> **Note**
>
> The VPC feature for automatically creating a DNS record is disabled for AWS IoT data and AWS IoT Credentials. To connect these endpoints, you must manually create a Private DNS record. For more information, see Private DNS for interface endpoints. For more information about AWS IoT Core VPC limitations, see Limitations of VPC endpoints.

Prerequisites

- You must install the AWS IoT Greengrass Core software using the manual provisioning steps. For more information, see Install AWS IoT Greengrass Core software with manual resource provisioning.

Limitations

- Operating a Greengrass core device in VPC is not supported in China Regions and AWS GovCloud (US) Regions.
- For more information about limitations of AWS IoT data and AWS IoT credential provider VPC endpoints, see Limitations.

Set up your Greengrass core device to operate in VPC

1. Get the AWS IoT endpoints for your AWS account, and save them to use later. Your device uses these endpoints to connect to AWS IoT. Do the following:

   a. Get the AWS IoT data endpoint for your AWS account.

   ```
   aws iot describe-endpoint --endpoint-type iot:Data-ATS
   ```

   The response looks similar to the following example, if the request succeeds.

   ```
   {
     "endpointAddress": "device-data-prefix-ats.iot.us-west-2.amazonaws.com"
   }
   ```

   b. Get the AWS IoT credentials endpoint for your AWS account.

   ```
   aws iot describe-endpoint --endpoint-type iot:CredentialProvider
   ```

   The response looks similar to the following example, if the request succeeds.

   ```
   {
     "endpointAddress": "device-credentials-prefix.credentials.iot.us-
   west-2.amazonaws.com"
   ```

```
}
```

2. Create an Amazon VPC interface for AWS IoT data and AWS IoT credentials endpoints:

   a. Navigate to the VPC **Endpoints** console, under **Virtual private cloud** on the left menu, choose **Endpoints** then **Create Endpoint**.

   b. In the **Create endpoint** page, specify the following information.

   - Choose **AWS services** for **Service category**.
   - For **Service Name**, search by entering the keyword `iot`. In the list of `iot` services displayed, choose the endpoint.

     If you create a VPC endpoint for AWS IoT Core data plane, choose the AWS IoT Core data plane API endpoint for your Region. The endpoint will be of the format `com.amazonaws.`*`region`*`.iot.data`.

     If you create a VPC endpoint for AWS IoT Core credential provider, choose the AWS IoT Core credential provider endpoint for your Region. The endpoint will be of the format `com.amazonaws.`*`region`*`.iot.credentials`.

     > **Note**
     >
     > The service name for AWS IoT Core data plane in China Region will be of the format `cn.com.amazonaws.`*`region`*`.iot.data`. Creating VPC endpoints for AWS IoT Core credential provider is not supported in China Region.

   - For **VPC** and **Subnets**, choose the VPC where you want to create the endpoint, and the Availability Zones (AZs) in which you want to create the endpoint network.
   - For **Enable DNS name**, make sure that **Enable for this endpoint** is not selected. Neither AWS IoT Core data plane nor AWS IoT Core credential provider supports private DNS names yet.
   - For **Security group**, choose the security groups you want to associate with the endpoint network interfaces.
   - Optionally, you can add or remove tags. Tags are name-value pairs that you use to associate with your endpoint.

   c. To create your VPC endpoint, choose **Create endpoint**.

3. After you create the AWS PrivateLink endpoint, in the **Details** tab of your endpoint, you'll see a list of DNS names. You can use one of these DNS names you created in this section to configure your private hosted zone.
4. Create an Amazon S3 endpoint. For more information, see Create a VPC endpoint for Amazon S3.
5. If you are using AWS-provided Greengrass components, additional endpoints and configurations may be required. To view the endpoints requirements, select the component from the list of AWS-provided components and look at the Requirements section. For example, the log manager component requirements advise that this component must be able to perform outbound requests to the endpoint `logs.region.amazonaws.com`.

   If you are using your own component, you may need to review the dependencies and perform additional testing to determine if any additional endpoints are required.
6. In Greengrass nucleus configuration, `greengrassDataPlaneEndpoint` must be set to **`iotdata`**. For more information, see Greengrass nucleus configuration.
7. If you are in the `us-east-1` region, set the configuration parameter `s3EndpointType` to **`REGIONAL`** in the Greengrass nucleus configuration. This feature is available for Greengrass nucleus versions 2.11.3 or later.

Example: Component configuration

```
{
"aws.greengrass.Nucleus": {
   "configuration": {
      "awsRegion": "us-east-1",
      "iotCredEndpoint": "xxxxxx.credentials.iot.region.amazonaws.com",
      "iotDataEndpoint": "xxxxxx-ats.iot.region.amazonaws.com",
      "greengrassDataPlaneEndpoint": "iotdata",
      "s3EndpointType": "REGIONAL"
      ...
     }
   }
}
```

The following table gives information about the corresponding custom private DNS aliases.

Service	VPC endpoint service name	VPC endpoint type	Custom private DNS alias	Notes
AWS IoT data	com.amazonaws. *region*.iot.d	Interface	*prefix*-ats.iot. *region*.a s.com	The private DNS record should match your account's AWS IoT data endpoint: aws iot describe-endpoint --endpoint-type iot:Data-ATS .
AWS IoT Credentials	com.amazonaws. *region*.iot.c entials	Interface	*prefix*.c: als.iot. s.com	The private DNS record should match your account AWS IoT Credentia

Service	VPC endpoint service name	VPC endpoint type	Custom private DNS alias	Notes
				ls endpoint: `aws iot describe-endpoint --endpoint-type iot:CredentialProvider .`
Amazon S3	`com.amazonaws.` *region* `.s3`	Interface		The DNS record is automatically created.

Security best practices for AWS IoT Greengrass

This topic contains security best practices for AWS IoT Greengrass.

Grant minimum possible permissions

Follow the principle of least privilege for your components by running them as unprivileged users. Components should not run as root unless it is absolutely necessary.

You can also run the AWS IoT Greengrass Core software itself as a non-root user to further limit privileges on your device. For more information, see Set up AWS IoT Greengrass V2 core devices as non-root.

Use the minimum set of permissions in IAM roles. Limit the use of the `*` wildcard for the `Action` and `Resource` properties in your IAM policies. Instead, declare a finite set of actions and resources

when possible. For more information about least privilege and other policy best practices, see the section called “Policy best practices”.

The least privilege best practice also applies to AWS IoT policies you attach to your Greengrass core.

Don't hardcode credentials in Greengrass components

Don't hardcode credentials in your user-defined Greengrass components. To better protect your credentials:

- To interact with AWS services, define permissions for specific actions and resources in the Greengrass core device service role.
- Use the secret manager component to store your credentials. Or, if the function uses the AWS SDK, use credentials from the default credential provider chain.

Don't log sensitive information

You should prevent the logging of credentials and other personally identifiable information (PII). We recommend that you implement the following safeguards even though access to local logs on a core device requires root privileges and access to CloudWatch Logs requires IAM permissions.

- Don't use sensitive information in MQTT topic paths.
- Don't use sensitive information in device (thing) names, types, and attributes in the AWS IoT Core registry.
- Don't log sensitive information in your user-defined Greengrass components or Lambda functions.
- Don't use sensitive information in the names and IDs of Greengrass resources:
  - Core devices
  - Components
  - Deployments
  - Loggers

Keep your device clock in sync

It's important to have an accurate time on your device. X.509 certificates have an expiry date and time. The clock on your device is used to verify that a server certificate is still valid. Device clocks can drift over time or batteries can get discharged.

For more information, see the Keep your device's clock in sync best practice in the *AWS IoT Core Developer Guide*.

Cipher Suite Recommendations

Greengrass default selects the latest TLS Cipher Suites available on the device. Consider disabling the use of legacy cipher suites on the device. For example, CBC cipher suites.

For more information, see the Java Cryptography Configuration.

See also

- Security best practices in AWS IoT Core in the *AWS IoT Developer Guide*
- Ten security golden rules for Industrial IoT solutions on the *Internet of Things on AWS Official Blog*

Run AWS IoT Greengrass V2 as a non-root user

Typically, the root user installs and runs the AWS IoT Greengrass Core software on Linux devices. To increase device security, you can set up a non-root user to run the AWS IoT Greengrass Core software instead. This section provides guidance on setting up non-root configurations.

Choose from the following topics based on your needs:

- **Set up AWS IoT Greengrass V2 core devices as non-root**

  If you are setting up new AWS IoT Greengrass V2 core devices and want to run them as a non-root user from the start, see Set up AWS IoT Greengrass V2 core devices as non-root. This guide covers multiple solutions based on your device constraints and security requirements.
- **Migrate AWS IoT Greengrass V2 core devices to non-root**

  If you have existing AWS IoT Greengrass core devices running as root and want to migrate them to a non-root configuration, see Migrate AWS IoT Greengrass V2 core devices to non-root.

Note

The non-root configurations in this section apply to Linux devices only. On Windows, AWS IoT Greengrass must run as a system service.

Topics

- Set up AWS IoT Greengrass V2 core devices as non-root
- Migrate AWS IoT Greengrass V2 core devices to non-root

Set up AWS IoT Greengrass V2 core devices as non-root

This page presents four solutions for running the AWS IoT Greengrass Core software as non-root. Review the comparison table to understand the features and tradeoffs of each solution, then use the decision flowchart to identify which one fits your requirements.

Note

The non-root solutions on this page apply to the AWS IoT Greengrass nucleus on Linux devices only. Windows is not included because AWS IoT Greengrass V2 must run as a system service on Windows. For standard root installation on Linux, see Install AWS IoT Greengrass Core software with automatic resource provisioning.
To run AWS IoT Greengrass nucleus lite as a non-root user, see Using Podman in the AWS IoT Greengrass nucleus lite GitHub repository.

Topics

- Choose a non-root solution
- Solution 1: Set up AWS IoT Greengrass V2 without root access
- Solution 2: Set up AWS IoT Greengrass V2 as non-root without component user separation
- Solution 3: Set up AWS IoT Greengrass V2 as non-root with component user separation
- Solution 4: Set up AWS IoT Greengrass V2 as root with limited capabilities
- Linux capabilities required by AWS IoT Greengrass V2

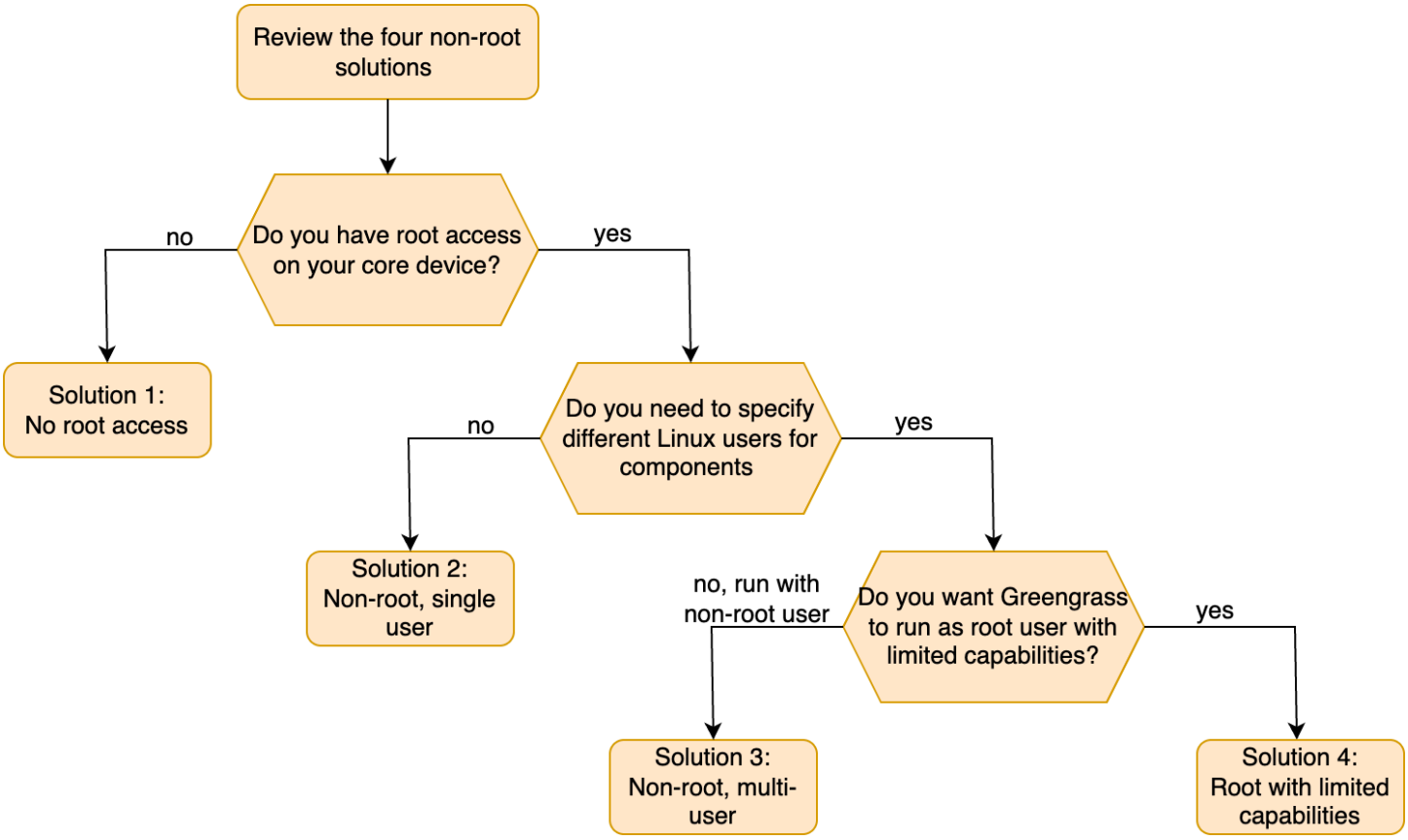
Choose a non-root solution

Use the following table to compare the non-root solutions and understand their tradeoffs. Each solution offers different capabilities depending on your security requirements and device constraints.

Non-root solutions

Solution	Requires root access	Can run components as different users	Runs Greengrass as system service	Best for
Solution 1: No root access	No	No	No (user service optional)	Devices where you don't have root access
Solution 2: Non-root, single user	Yes (setup only)	No	Yes	Running Greengrass as a non-root user with all components running as the same user
Solution 3: Non-root, multi-user	Yes (setup only)	Yes	Yes	Running Greengrass as a non-root user while running components as different users
Solution 4: Root with limited capabilities	Yes	Yes	Yes	Running Greengrass as root with a limited set of Linux capabilities

The following flowchart guides you through selecting the appropriate solution based on your device constraints and requirements.



Solution 1: Set up AWS IoT Greengrass V2 without root access

Use this solution when you don't have root access on the device. In this configuration, the AWS IoT Greengrass Core software runs entirely as a non-root user without elevated privileges.

Tradeoffs

This solution has the following limitations:

- **No component user separation** – All components run as the same user that runs the AWS IoT Greengrass Core software. You cannot use the `posixUser` configuration to run components as different users.
- **RequiresPrivilege ignored** – The AWS IoT Greengrass Core software ignores the `RequiresPrivilege` option in component recipes. Components cannot request elevated privileges.
- **No system service** – You cannot install the AWS IoT Greengrass Core software as a system service. You can optionally configure AWS IoT Greengrass V2 to run as a systemd user service.

Prerequisites

This solution requires:

- A non-root user account on the device
- Write access to the directory where you want to install the AWS IoT Greengrass Core software

To install and run AWS IoT Greengrass V2 without root access

1. Complete the following steps from Install AWS IoT Greengrass Core software with automatic resource provisioning: set up your device environment, provide credentials, and download the AWS IoT Greengrass Core software.
2. Create the installation directory and ensure your user owns it.

   ```
   mkdir -p $HOME/greengrass/v2
   ```

3. Run the installer without sudo. Set `--component-default-user` to your current user.

   ```
   java -Droot="$HOME/greengrass/v2" -Dlog.store=FILE \
     -jar ./GreengrassInstaller/lib/Greengrass.jar \
     --aws-region region \
     --thing-name MyGreengrassCore \
     --thing-group-name MyGreengrassCoreGroup \
     --thing-policy-name GreengrassV2IoTThingPolicy \
     --tes-role-name GreengrassV2TokenExchangeRole \
     --tes-role-alias-name GreengrassCoreTokenExchangeRoleAlias \
     --component-default-user $USER \
     --provision true
   ```

   Do not use `--setup-system-service true` because you don't have root access to create a system service.

(Optional) Set up a systemd user service

You can configure a systemd user service to manage the AWS IoT Greengrass Core software. This allows the software to start automatically when you log in.

To set up a systemd user service

1. Stop the AWS IoT Greengrass Core software if it is currently running.

```
kill $(cat $HOME/greengrass/v2/alts/loader.pid)
```

2. Create the systemd user service directory.

```
mkdir -p $HOME/.config/systemd/user
```

3. Create the service file at `$HOME/.config/systemd/user/greengrass.service` with the following content.

```
[Unit]
Description=Greengrass Core

[Service]
Type=simple
PIDFile=%h/greengrass/v2/alts/loader.pid
RemainAfterExit=no
Restart=on-failure
RestartSec=10
ExecStart=/bin/sh %h/greengrass/v2/alts/current/distro/bin/loader
Environment="JAVA_HOME=/path/to/java"

[Install]
WantedBy=default.target
```

   In systemd user unit files, `%h` is a specifier that resolves to the home directory of the user running the service.

   Replace */path/to/java* with the path to your Java installation.

4. Enable and start the service.

```
systemctl --user daemon-reload
systemctl --user enable greengrass.service
systemctl --user start greengrass.service
```

Restart and OTA update behavior

The behavior depends on whether you configured a systemd user service.

With a user service

- **Device reboot** – The AWS IoT Greengrass Core software starts automatically when the user logs in.
- **OTA update** – OTA updates succeed and the software restarts automatically.

Without a user service

- **Device reboot** – The AWS IoT Greengrass Core software does not restart automatically. You must manually start it.
- **OTA update** – OTA updates succeed, but you must manually start the AWS IoT Greengrass Core software afterward.

System resource limits

Per-component resource limits using cgroups do not work in this solution because non-root users cannot create cgroup directories in `/sys/fs/cgroup/`. If you deploy a component with resource limits configured, the AWS IoT Greengrass Core software ignores the configured limits.

You can use the following alternatives to manage resource usage:

- **Systemd user service limits** – If you run the AWS IoT Greengrass Core software as a systemd user service, you can add resource limits to the service file at `$HOME/.config/systemd/user/greengrass.service`. These limits apply to the entire service, including the Greengrass nucleus and all components.

  ```
  MemoryMax=2G
  CPUQuota=100%
  ```

- **Greengrass nucleus JVM limits** – You can limit the Greengrass nucleus process memory by configuring JVM options. For more information, see Configure the AWS IoT Greengrass Core software.

Solution 2: Set up AWS IoT Greengrass V2 as non-root without component user separation

Use this solution when you have root access for initial setup and want the AWS IoT Greengrass Core software to run as a non-root system service, but you don't need to run components as different users. This is similar to Solution 1, but the software runs as a system service that starts automatically on boot.

Tradeoffs

This solution has the following limitations:

- **No component user separation** – All components run as the same user that runs the AWS IoT Greengrass Core software. You cannot use the `posixUser` configuration to run components as different users.
- **RequiresPrivilege ignored** – The AWS IoT Greengrass Core software ignores the `RequiresPrivilege` option in component recipes. Components cannot request elevated privileges.

Prerequisites

This solution requires:

- Root access for initial setup
- systemd on your device

To install and run AWS IoT Greengrass V2 as non-root without component user separation

1. Complete the following steps from Install AWS IoT Greengrass Core software with automatic resource provisioning: set up your device environment, provide credentials, and download the AWS IoT Greengrass Core software.
2. Create a non-root user that will run the AWS IoT Greengrass Core software and all components.

   ```
   sudo useradd --create-home gg_non_root
   ```

3. Log in as the non-root user and run the installer. Set `--component-default-user` to the same user. Do not use `sudo` and do not set up a system service.

   ```
   java -Droot="/home/gg_non_root/greengrass/v2" -Dlog.store=FILE \
     -jar ./GreengrassInstaller/lib/Greengrass.jar \
     --aws-region region \
     --thing-name MyGreengrassCore \
     --thing-group-name MyGreengrassCoreGroup \
     --thing-policy-name GreengrassV2IoTThingPolicy \
     --tes-role-name GreengrassV2TokenExchangeRole \
     --tes-role-alias-name GreengrassCoreTokenExchangeRoleAlias \
   ```

```
  --component-default-user gg_non_root \
  --provision true \
  --setup-system-service false
```

4. As root, create the system service file at `/etc/systemd/system/greengrass.service` with the following content.

```
[Unit]
Description=Greengrass Core
After=network.target

[Service]
Type=simple
User=gg_non_root
PIDFile=/home/gg_non_root/greengrass/v2/alts/loader.pid
RemainAfterExit=no
Restart=on-failure
RestartSec=10
ExecStart=/bin/sh -c "exec /home/gg_non_root/greengrass/v2/alts/current/distro/bin/
loader >> /home/gg_non_root/greengrass/v2/logs/loader.log 2>&1"
KillMode=mixed
NoNewPrivileges=true
ProtectSystem=strict
ReadWritePaths=/home/gg_non_root/greengrass /tmp

[Install]
WantedBy=multi-user.target
```

5. Stop the running AWS IoT Greengrass Core instance, then enable and start the system service.

```
sudo systemctl daemon-reload
sudo systemctl enable greengrass.service
sudo systemctl start greengrass.service
```

Restart and OTA update behavior

In this solution:

- The AWS IoT Greengrass Core software runs as a system service and restarts automatically on failure or device reboot.

- OTA updates to the AWS IoT Greengrass Core software work. The service restarts automatically as the configured non-root user.

System resource limits

Per-component resource limits using cgroups do not work in this solution because non-root users without Linux capabilities cannot create cgroup directories in `/sys/fs/cgroup/`. If you deploy a component with resource limits configured, the AWS IoT Greengrass Core software ignores the configured limits.

You can use the following alternatives to manage resource usage:

- **Systemd service limits** – You can add resource limits to the system service file at `/etc/systemd/system/greengrass.service`. These limits apply to the entire service, including the Greengrass nucleus and all components.

  ```
  MemoryMax=2G
  CPUQuota=100%
  ```

- **Greengrass nucleus JVM limits** – You can limit the Greengrass nucleus process memory by configuring JVM options. For more information, see Configure the AWS IoT Greengrass Core software.

Solution 3: Set up AWS IoT Greengrass V2 as non-root with component user separation

Use this solution when you have root access for initial setup but want the AWS IoT Greengrass Core software to run as a non-root user while maintaining the ability to run components as different users. This configuration uses Linux capabilities and sudoers to allow the non-root user to switch to other users when running components.

Tradeoffs

This solution has the following limitations:

- **Requires sudoers configuration** – You must configure sudoers to allow the AWS IoT Greengrass V2 user to run commands as other users.

Prerequisites

This solution requires:

- Root access for initial setup
- systemd version 229 or later, which supports `AmbientCapabilities`. To check your version, run `systemctl --version`.

To install and run AWS IoT Greengrass V2 as non-root with component user separation

1. Complete the following steps from Install AWS IoT Greengrass Core software with automatic resource provisioning: set up your device environment, provide credentials, and download the AWS IoT Greengrass Core software.
2. Create a non-root user that will run the AWS IoT Greengrass Core software.

   ```
   sudo useradd --create-home gg_non_root
   ```

3. Add the non-root user to sudoers so that it can run commands as the component user. Create a file at `/etc/sudoers.d/gg_non_root`.

   ```
   gg_non_root ALL=(ggc_user:ggc_group) NOPASSWD: SETENV: /bin/sh, /bin/bash
   ```

   If you configure components to run as additional users with `posixUser`, add a sudoers entry for each user. For example:

   ```
   gg_non_root ALL=(ggc_user:ggc_group) NOPASSWD: SETENV: /bin/sh, /bin/bash
   gg_non_root ALL=(another_user:another_group) NOPASSWD: SETENV: /bin/sh, /bin/bash
   ```

4. Log in as the non-root user and run the installer. Do not use `sudo` and do not set up a system service.

   ```
   java -Droot="/home/gg_non_root/greengrass/v2" -Dlog.store=FILE \
     -jar ./GreengrassInstaller/lib/Greengrass.jar \
     --aws-region region \
     --thing-name MyGreengrassCore \
     --thing-group-name MyGreengrassCoreGroup \
     --thing-policy-name GreengrassV2IoTThingPolicy \
     --tes-role-name GreengrassV2TokenExchangeRole \
     --tes-role-alias-name GreengrassCoreTokenExchangeRoleAlias \
     --component-default-user ggc_user:ggc_group \
     --provision true \
   ```

```
  --setup-system-service false
```

5. As root, create the system service file at `/etc/systemd/system/greengrass.service` with the following content.

```
[Unit]
Description=Greengrass Core
After=network.target

[Service]
Type=simple
User=gg_non_root
PIDFile=/home/gg_non_root/greengrass/v2/alts/loader.pid
RemainAfterExit=no
Restart=on-failure
RestartSec=10
ExecStart=/bin/sh -c "exec /home/gg_non_root/greengrass/v2/alts/current/distro/bin/
loader >> /home/gg_non_root/greengrass/v2/logs/loader.log 2>&1"
KillMode=mixed
AmbientCapabilities=CAP_CHOWN CAP_DAC_OVERRIDE CAP_DAC_READ_SEARCH CAP_FOWNER
 CAP_SETUID CAP_SETGID CAP_SYS_RESOURCE CAP_AUDIT_WRITE CAP_SYS_BOOT
CapabilityBoundingSet=CAP_CHOWN CAP_DAC_OVERRIDE CAP_DAC_READ_SEARCH CAP_FOWNER
 CAP_SETUID CAP_SETGID CAP_SYS_RESOURCE CAP_AUDIT_WRITE CAP_SYS_BOOT
ProtectSystem=strict
ReadWritePaths=/home/gg_non_root/greengrass /tmp

[Install]
WantedBy=multi-user.target
```

For information about the required capabilities, see [Linux capabilities required by AWS IoT Greengrass V2].

- `AmbientCapabilities` grants the specified Linux capabilities to the non-root user running the service. This allows the AWS IoT Greengrass Core software to perform privileged operations such as switching users when running components, without running as root.
- `CapabilityBoundingSet` limits the maximum set of capabilities that the service and its child processes can use. Capabilities not in this set are permanently dropped.
- `ProtectSystem=strict` makes the entire file system read-only for the service, preventing modification of the operating system.
- `ReadWritePaths` specifies the only directories that the service can write to.

6. Stop the running AWS IoT Greengrass Core instance, then enable and start the system service.

```
sudo systemctl daemon-reload
sudo systemctl enable greengrass.service
sudo systemctl start greengrass.service
```

Restart and OTA update behavior

In this solution:

- The AWS IoT Greengrass Core software runs as a system service and restarts automatically on failure or device reboot.
- OTA updates to the AWS IoT Greengrass Core software work. The service restarts automatically as the configured non-root user.

System resource limits

Per-component resource limits for memory and CPU work in this solution the same way as when running the AWS IoT Greengrass Core software as root.

Solution 4: Set up AWS IoT Greengrass V2 as root with limited capabilities

Use this solution when you want the AWS IoT Greengrass Core software to run as root but with a reduced set of Linux capabilities. This configuration provides the full functionality of running as root while limiting the attack surface by restricting the capabilities available to the software and its components.

Tradeoffs

This solution has the following considerations:

- **Components with RequiresPrivilege have limited capabilities** – Components that use `RequiresPrivilege` run with the same limited set of capabilities as the AWS IoT Greengrass Core software, not full root privileges.

Prerequisites

This solution requires:

- Root access
- systemd on your device

To install and run AWS IoT Greengrass V2 as root with limited capabilities

1. Complete the following steps from Install AWS IoT Greengrass Core software with automatic resource provisioning: set up your device environment, provide credentials, and download the AWS IoT Greengrass Core software.
2. Modify the service template file at `GreengrassInstaller/bin/greengrass.service.template`, in the directory where you downloaded the AWS IoT Greengrass Core software. Add the following lines to the `[Service]` section before you run the installer:

   ```
   CapabilityBoundingSet=CAP_CHOWN CAP_DAC_OVERRIDE CAP_DAC_READ_SEARCH CAP_FOWNER
    CAP_SETUID CAP_SETGID CAP_SYS_RESOURCE CAP_AUDIT_WRITE CAP_SYS_BOOT
   ProtectSystem=strict
   ReadWritePaths=/greengrass /tmp
   ```

   - `CapabilityBoundingSet` is a systemd security feature that limits the Linux capabilities available to the AWS IoT Greengrass Core software and all of its child processes. By configuring a bounding set, you restrict what the Greengrass nucleus process and components can do, even when running as root. For information about each capability, see Linux capabilities required by AWS IoT Greengrass V2.
   - `ProtectSystem=strict` makes the entire file system read-only for the service, preventing modification of the operating system. This provides systemd sandboxing that protects system files even if a malicious component runs with elevated privileges.
   - `ReadWritePaths` specifies the only directories that the service can write to. Together with `ProtectSystem=strict`, this limits the service to writing only to the AWS IoT Greengrass V2 root directory and `/tmp`.
3. Run the installer with `--setup-system-service true`.

   ```
   sudo -E java -Droot="/greengrass/v2" -Dlog.store=FILE \
     -jar ./GreengrassInstaller/lib/Greengrass.jar \
     --aws-region region \
     --thing-name MyGreengrassCore \
     --thing-group-name MyGreengrassCoreGroup \
     --thing-policy-name GreengrassV2IoTThingPolicy \
   ```

```
  --tes-role-name GreengrassV2TokenExchangeRole \
  --tes-role-alias-name GreengrassCoreTokenExchangeRoleAlias \
  --component-default-user ggc_user:ggc_group \
  --provision true \
  --setup-system-service true
```

Restart and OTA update behavior

In this solution:

- The AWS IoT Greengrass Core software runs as a system service and restarts automatically on failure or device reboot.
- OTA updates to the AWS IoT Greengrass Core software work. The software updates and restarts automatically.

System resource limits

Per-component resource limits for memory and CPU work in this solution the same way as when running the AWS IoT Greengrass Core software as root.

Linux capabilities required by AWS IoT Greengrass V2

The following table describes the Linux capabilities required by the AWS IoT Greengrass Core software when running in non-root configurations. These capabilities are used in Solutions 3 and 4.

Required Linux capabilities

Capability	Description	Required for
`CAP_CHOWN`	Make arbitrary changes to file UIDs and GIDs	Changing file ownership depending on the user executing the component
`CAP_DAC_OVERRIDE`	Bypass file read, write, and execute permission checks	Allowing the Greengrass nucleus user to execute files when used for scripts with `RequiresPrivilege`

Capability	Description	Required for
CAP_DAC_READ_SEARCH	Bypass file read permission checks and directory read and execute permission checks	Walking the folder hierarchy even for folders the Greengrass nucleus user does not have permission to read
CAP_FOWNER	Bypass permission checks on operations that normally require the filesystem UID of the process to match the UID of the file	Bypassing file ownership checks
CAP_SETUID	Make arbitrary manipulations of process UIDs	Using sudo when executing scripts as a different user than the user running the Greengrass nucleus
CAP_SETGID	Make arbitrary manipulations of process GIDs	Using sudo when executing scripts as a different group than the group running the Greengrass nucleus
CAP_SYS_RESOURCE	Override resource limits	Setting resource limits on component processes, even if limits are not specified in deployments
CAP_AUDIT_WRITE	Write records to kernel auditing log	Allowing sudo to write to the kernel audit log
CAP_SYS_BOOT	Use reboot(2) and kexec_load(2)	Allowing the AWS IoT Greengrass Core software to reboot the device when components manage system restarts

Migrate AWS IoT Greengrass V2 core devices to non-root

> **Important**
>
> This topic describes a recommended migration approach. You might need to adapt these steps for your device configuration, operating system, and installed software.

By default, AWS IoT Greengrass runs as the root user on Linux devices. You can improve the security posture of your devices by migrating AWS IoT Greengrass core devices to run as a non-root user with only the Linux capabilities that AWS IoT Greengrass requires. You can migrate a single device or scale the migration across a fleet.

Considerations

Consider the following before you migrate to non-root:

- This migration procedure applies to AWS IoT Greengrass core devices running the AWS IoT Greengrass nucleus (Java-based) on Linux with systemd. It does not apply to devices running AWS IoT Greengrass nucleus lite.
- Test the migration on a single device in a development environment before you migrate your production fleet.

Migrate a single device

This migration procedure follows the approach described in Solution 3: Set up AWS IoT Greengrass V2 as non-root with component user separation. Before you begin, complete the prerequisites in that topic, including the sudoers configuration that allows the non-root user to run components as separate users.

Use this procedure to migrate a single device before you migrate your fleet.

Step 1: Create the non-root user and group

Connect to your device and create the system user and group that AWS IoT Greengrass runs as. Replace *user-name* and *group-name* with the names that you want to use.

```
sudo groupadd --system group-name
sudo useradd --system --create-home --shell /sbin/nologin -g group-name user-name
```

Step 2: Back up the file ownership

Stop AWS IoT Greengrass and save the current file ownership structure so that you can restore it if you need to roll back.

```
sudo systemctl stop greengrass.service
sudo mkdir -p /opt/greengrass-backup
sudo find /greengrass/v2 -exec stat -c 'chown %U:%G "%n"' {} \; | sudo tee /opt/
greengrass-backup/restore_ownership.sh > /dev/null
sudo chmod 700 /opt/greengrass-backup/restore_ownership.sh
```

> **Note**
>
> The commands in this topic use `/greengrass/v2` as the AWS IoT Greengrass root path. If you installed AWS IoT Greengrass to a different location, replace `/greengrass/v2` with your actual AWS IoT Greengrass root path.

Step 3: Change file ownership

Transfer ownership of the AWS IoT Greengrass directory to the non-root user.

```
sudo chown -R user-name:group-name /greengrass/v2
```

Step 4: Create the systemd drop-in override

Create a drop-in directory and configuration file that tells systemd to run AWS IoT Greengrass as the non-root user with the required Linux capabilities.

```
sudo mkdir -p /etc/systemd/system/greengrass.service.d
```

The migration uses a systemd drop-in override file at `/etc/systemd/system/greengrass.service.d/10-nonroot.conf`. A drop-in override replaces settings in the base service file without modifying it directly. This override configures systemd to run the AWS IoT Greengrass service as the non-root user with a limited set of Linux capabilities. Removing the drop-in directory causes systemd to revert to the base `greengrass.service` file, which runs as root with the original configuration.

Create the file `/etc/systemd/system/greengrass.service.d/10-nonroot.conf` with the following content:

```
[Service]
User=user-name
AmbientCapabilities=CAP_CHOWN CAP_DAC_OVERRIDE CAP_DAC_READ_SEARCH CAP_FOWNER
 CAP_SETUID CAP_SETGID CAP_SYS_RESOURCE CAP_AUDIT_WRITE
CapabilityBoundingSet=CAP_CHOWN CAP_DAC_OVERRIDE CAP_DAC_READ_SEARCH CAP_FOWNER
 CAP_SETUID CAP_SETGID CAP_SYS_RESOURCE CAP_AUDIT_WRITE
ProtectSystem=strict
ReadWritePaths=/greengrass/v2 /tmp
```

For information about these systemd directives and the required capabilities, see Solution 3: Set up AWS IoT Greengrass V2 as non-root with component user separation.

Step 5: Reload systemd and start AWS IoT Greengrass

Run the following commands to reload the systemd daemon and start AWS IoT Greengrass.

```
sudo systemctl daemon-reload
sudo systemctl start greengrass.service
```

Step 6: Verify the migration

Confirm that AWS IoT Greengrass is running as the non-root user.

```
sudo systemctl status greengrass.service
ps -ef | grep greengrass
```

The output shows the non-root user as the process owner instead of `root`.

Next, confirm that all deployed components are running. You must deploy the `aws.greengrass.Cli` component on the device to run this command.

```
sudo /greengrass/v2/bin/greengrass-cli component list
```

Each component should report a `State` of `RUNNING` or `FINISHED`. `FINISHED` is the expected terminal state for `aws.greengrass.Nucleus`. Investigate any component in the `BROKEN` or `ERRORED` state before you consider the migration complete.

Roll back a single device

If you need to restore AWS IoT Greengrass to run as root, remove the drop-in override, restore file ownership, and restart AWS IoT Greengrass.

```
sudo systemctl stop greengrass.service
sudo rm -rf /etc/systemd/system/greengrass.service.d
sudo bash /opt/greengrass-backup/restore_ownership.sh
sudo systemctl daemon-reload
sudo systemctl start greengrass.service
```

Run the verification commands from Step 6: Verify the migration again to confirm that AWS IoT Greengrass is running as root. The process should show `root` as the owner.

After you confirm the rollback succeeded, you can optionally clean up the migration artifacts:

- **Backup directory** — You can keep `/opt/greengrass-backup` if you plan to retry the migration later, or remove it:

  ```
  sudo rm -r /opt/greengrass-backup
  ```

- **Non-root user and group** — If you no longer plan to run AWS IoT Greengrass as a non-root user, you can remove the user and group that you created in Step 1:

  ```
  sudo userdel user-name
  sudo groupdel group-name
  ```

  If other services or components use this user, do not remove the user.

Migrate a fleet of devices

After you verify the migration on a single device, you can scale the migration across your fleet. Choose one of the following methods:

- Use a custom AWS IoT Greengrass bootstrap component
- Use AWS Systems Manager

Use a custom AWS IoT Greengrass bootstrap component

You can create a custom AWS IoT Greengrass component that automates the single-device migration steps across your fleet. This approach is self-contained and uses only AWS IoT Greengrass deployments. It does not require external tools such as Systems Manager.

The component uses the Bootstrap lifecycle with `RequiresPrivilege` set to `true`, which runs the component as root during deployment. The bootstrap script performs the same privileged operations as the single-device procedure:

- Create the non-root user and group if they do not already exist.
- Back up the file ownership structure for rollback.
- Change ownership of the AWS IoT Greengrass root directory to the non-root user.
- Create the systemd drop-in override file.
- Reload the systemd daemon.
- Exit with code 101 to reboot the device.

> **Important**
>
> The component must exit with code 101, not 100. Exit code 100 restarts only the AWS IoT Greengrass JVM process, which uses the systemd service configuration already cached in memory. A full reboot forces systemd to read the new drop-in override from disk and start AWS IoT Greengrass as the non-root user. The reboot temporarily disrupts all services on the device.

Make the component idempotent so that redeployments do not fail if you already migrated the device.

After the migration deployment completes, verify that your devices are running as the non-root user. For more information, see Step 6: Verify the migration. You can deploy a verification component that checks the running user and reports the result, or monitor the deployment status in the AWS IoT Greengrass console.

Set up a rollback mechanism

To restore the root configuration remotely across your fleet, your migration component must set up a rollback mechanism. One approach is to use systemd to monitor for a trigger file. This approach requires the following systemd resources:

- A *rollback script* that removes the drop-in override, restores the original file ownership from the backup, reloads the systemd daemon, and restarts AWS IoT Greengrass as root.

- A systemd *oneshot service* that runs the rollback script as root. A oneshot service runs a single task and then exits, rather than running as a long-lived daemon.
- A systemd *path unit* that monitors a specific file path, such as `/greengrass/v2/rollback/.rollback-trigger`. When the file appears, the path unit automatically starts the oneshot rollback service.

With this mechanism in place, you can roll back a device by deploying a simple AWS IoT Greengrass component that creates the trigger file. The systemd path unit detects the file, runs the rollback service as root, and restores AWS IoT Greengrass to its original root configuration.

Use AWS Systems Manager

If you use Systems Manager to manage your devices, you can create an SSM document that automates the single-device migration steps. Because SSM runs independently of AWS IoT Greengrass, it can stop and restart the AWS IoT Greengrass service directly without rebooting the device.

Your SSM document must meet the following requirements:

- Stop AWS IoT Greengrass and back up the current service configuration and file ownership structure.
- Create the non-root user and group.
- Change ownership of the AWS IoT Greengrass directory to the non-root user.
- Create the systemd drop-in override file.
- Reload the systemd daemon and start AWS IoT Greengrass.
- Verify that AWS IoT Greengrass started successfully as the non-root user.

To run the migration across your fleet, use Systems Manager Run Command to execute the SSM document against your devices using tags or resource groups. You can monitor the command execution status in the Systems Manager console, which shows the result for each target device, including any errors.

To roll back, create a separate SSM document that stops AWS IoT Greengrass, removes the drop-in override, restores the original file ownership, reloads the systemd daemon, and restarts AWS IoT Greengrass as root. Run this document against the devices that you need to roll back.

For more information about running commands on managed nodes, see Running commands using Systems Manager Run Command.

Using AWS IoT Device Tester for AWS IoT Greengrass V2

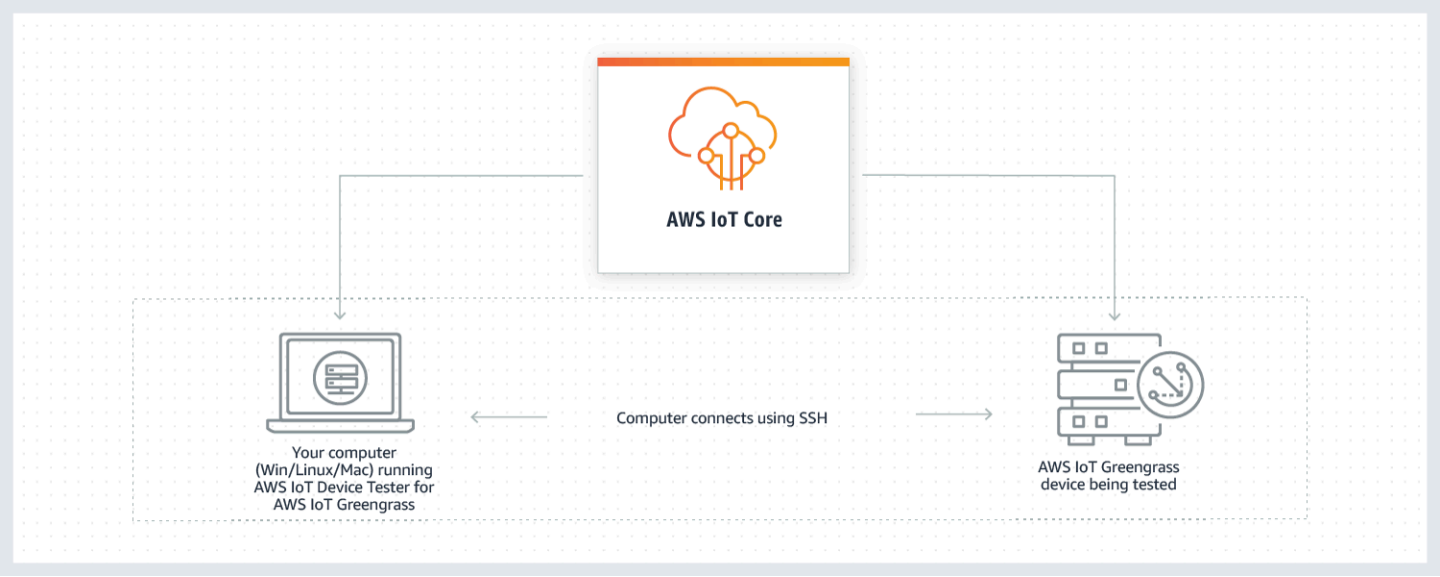
AWS IoT Device Tester (IDT) is a downloadable testing framework that lets you validate IoT devices. You can use IDT for AWS IoT Greengrass to run the AWS IoT Greengrass qualification suite, and create and run custom test suites for your devices.

IDT for AWS IoT Greengrass runs on your host computer (Windows, macOS, or Linux) connected to the device to be tested. It runs tests and aggregates results. It also provides a command line interface to manage the testing process.

AWS IoT Greengrass qualification suite

Use AWS IoT Device Tester for AWS IoT Greengrass V2 to verify that the AWS IoT Greengrass Core software runs on your hardware and can communicate with the AWS Cloud. It also performs end-to-end tests with AWS IoT Core. For example, it verifies that your device can deploy components and upgrade them.

If you want to add your hardware to the AWS Partner Device Catalog, run the AWS IoT Greengrass qualification suite to generate test reports that you can submit to AWS IoT. For more information, see AWS Device Qualification Program.



IDT for AWS IoT Greengrass V2 organizes tests using the concepts of *test suites* and *test groups*.

- A test suite is the set of test groups used to verify that a device works with particular versions of AWS IoT Greengrass.

- A test group is the set of individual tests related to a particular feature, such as component deployments.

For more information, see Use IDT to run the AWS IoT Greengrass qualification suite.

Custom test suites

Starting in IDT v4.0.1, IDT for AWS IoT Greengrass V2 combines a standardized configuration setup and result format with a test suite environment that enables you to develop custom test suites for your devices and device software. You can add custom tests for your own internal validation or provide them to your customers for device verification.

How a test writer configures a custom test suite determines the settings configurations that are required to run custom test suites. For more information, see Use IDT to develop and run your own test suites.

Supported versions of AWS IoT Device Tester for AWS IoT Greengrass V2

This topic lists supported versions of IDT for AWS IoT Greengrass V2. As a best practice, we recommend that you use the latest version of IDT for AWS IoT Greengrass V2 that supports your target version of AWS IoT Greengrass V2. New releases of AWS IoT Greengrass might require you to download a new version of IDT for AWS IoT Greengrass V2. You receive a notification when you start a test run if IDT for AWS IoT Greengrass V2 is not compatible with the version of AWS IoT Greengrass you are using.

By downloading the software, you agree to the AWS IoT Device Tester License Agreement.

> **Note**
>
> IDT does not support being run by multiple users from a shared location, such as an NFS directory or a Windows network shared folder. We recommend that you extract the IDT package to a local drive and run the IDT binary on your local workstation.

Latest IDT version for AWS IoT Greengrass V2

You can use this version of IDT for AWS IoT Greengrass V2 with the AWS IoT Greengrass version listed here.

IDT v4.9.4 for AWS IoT Greengrass

Supported AWS IoT Greengrass versions:

- Greengrass nucleus v2.12.0, v2.11.0, v2.10.0, and v2.9.5

IDT software downloads:

- IDT v4.9.4 with test suite GGV2Q_2.5.4 for Linux
- IDT v4.9.4 with test suite GGV2Q_2.5.4 for macOS
- IDT v4.9.4 with test suite GGV2Q_2.5.4 for Windows

Release notes:

- Enables device validation and qualification for devices running AWS IoT Greengrass Core software versions 2.12.0, 2.11.0, 2.10.0, and 2.9.5.
- Removes stream manager and machine learning test groups.

Additional notes:

- If your device uses a HSM and you are using nucleus 2.10.x, migrate to Greengrass nucleus version 2.11.0 or later.

Test suite version:

`GGV2Q_2.5.4`

- Released 2024.05.03

Earlier IDT versions for AWS IoT Greengrass

The following earlier versions of IDT for AWS IoT Greengrass V2 are also supported.

IDT v4.9.3 for AWS IoT Greengrass

Supported AWS IoT Greengrass versions:

- Greengrass nucleus v2.12.0, v2.11.0, v2.10.0, and v2.9.5

IDT software downloads:

- IDT v4.9.3 with test suite GGV2Q_2.5.3 for Linux

- IDT v4.9.3 with test suite GGV2Q_2.5.3 for macOS
- IDT v4.9.3 with test suite GGV2Q_2.5.3 for Windows

Release notes:

- Fixes an issue in the component tests when testing a Linux device from a Windows host or vice versa.
- Removes the `localcomponent` test case from the `component` test group. This test case is no longer required for qualification.

Additional notes:

- If your device uses a HSM and you are using nucleus 2.10.x, migrate to Greengrass nucleus version 2.11.0 or later.

Test suite version:

`GGV2Q_2.5.3`

- Released 2024.04.05

Unsupported versions of AWS IoT Device Tester for AWS IoT Greengrass V2

This topic lists unsupported versions of IDT for AWS IoT Greengrass V2. Unsupported versions do not receive bug fixes or updates. For more information, see the section called "Support policy for AWS IoT Device Tester for AWS IoT Greengrass".

IDT v4.9.2 for AWS IoT Greengrass

Release notes:

- Fixes an issue where the Lambda test suite fails due to Java 8 being deprecated.

Test suite version:

`GGV2Q_2.5.2`

- Released 2024.03.18

IDT v4.9.1 for AWS IoT Greengrass

Release notes:

- Enables you to validate and qualify devices running AWS IoT Greengrass Core software versions 2.12.0, 2.11.0, 2.10.0, and 2.9.5.

- Minor bug fixes.

Test suite version:

`GGV2Q_2.5.1`

- Released 2023.10.05

IDT v4.7.0 for AWS IoT Greengrass

Supported AWS IoT Greengrass versions:

- Greengrass nucleus v2.11.0, v2.10.0, and v2.9.5

Release notes:

- Enables you to validate and qualify devices running AWS IoT Greengrass Core software versions 2.11.0, 2.10.0, and 2.9.5.
- Adds support to store IDT userdata values in AWS Systems Manager Parameter Store and fetch them into configuration using placeholder syntax.
- Minor bug fixes.

Test suite version:

`GGV2Q_2.5.0`

- Released 2022.12.13

IDT v4.5.11 for AWS IoT Greengrass

Release notes:

- Enables you to validate and qualify devices running AWS IoT Greengrass Core software versions 2.9.1, 2.9.0, 2.8.1, 2.8.0, 2.7.0, and 2.6.0.
- Adds support to test PreInstalled Greengrass on a core device.
- Minor bug fixes.

Test suite version:

`GGV2Q_2.4.1`

- Released 2022.10.13

IDT v4.5.8 for AWS IoT Greengrass

Release notes:

- Enables you to validate and qualify devices running AWS IoT Greengrass Core software versions 2.7.0, 2.6.0, and 2.5.6.
- Enables you to test with PreInstalled Greengrass on a core device.

- Minor bug fixes.

Test suite version:

`GGV2Q_2.4.0`

- Released 2022.08.12

IDT v4.5.3 for AWS IoT Greengrass

Release notes:

- Enables you to validate and qualify devices running AWS IoT Greengrass Core software versions 2.7.0, 2.6.0, 2.5.6, 2.5.5, 2.5.4, and 2.5.3.
- Updates DockerApplicationManager test to use an ECR-based docker image.
- Minor bug fixes.

Test suite version:

`GGV2Q_2.3.1`

- Released 2022.04.15

IDT v4.5.1 for AWS IoT Greengrass

Release notes:

- Enables you to validate and qualify devices running AWS IoT Greengrass Core software v2.5.3.
- Adds support for validating and qualifying Linux-based devices that use a hardware security module (HSM) to store the private key and certificate that are used by AWS IoT Greengrass Core software.
- Implements the new IDT test orchestrator for configuring custom test suites. For more information, see Configure the IDT test orchestrator.
- Additional minor bug fixes.

Test suite version:

`GGV2Q_2.3.0`

- Released 2022.01.11

IDT v4.4.1 for AWS IoT Greengrass

Release notes:

- Enables you to validate and qualify devices running AWS IoT Greengrass Core software v2.5.2.

- Adds support for using a user-defined IAM role as the token exchange role that the device under test assumes to interact with AWS resources.

  You can specify the IAM role in the `userdata.json` file. If you specify a custom role, IDT uses that role instead of creating the default token exchange role during the test run.

- Additional minor bug fixes.

Test suite version:

`GGV2Q_2.2.1`

- Released 2021.12.12

IDT v4.4.0 for AWS IoT Greengrass

Release notes:

- Enables you to validate and qualify devices running AWS IoT Greengrass Core software v2.5.0.
- Adds support for validating and qualifying devices running AWS IoT Greengrass Core software on Windows.
- Supports the use of public key validation for secure shell (SSH) device connections.
- Improves the IDT permissions IAM policy with security best practices.
- Additional minor bug fixes.

Test suite version:

`GGV2Q_2.1.0`

- Released 2021.11.19

IDT v4.2.0 for AWS IoT Greengrass

Release notes:

- Includes support for qualification of the following features on devices running AWS IoT Greengrass Core software v2.2.0 and later versions:
  - Docker—Validates that devices can download a Docker container image from Amazon Elastic Container Registry (Amazon ECR).
  - Machine learning—Validates that devices can perform machine learning (ML) inference using the Deep Learning Runtime or TensorFlow Lite ML frameworks.
  - Stream manager—Validates that devices can download, install, and run the AWS IoT Greengrass stream manager.

- Enables you to validate and qualify devices running AWS IoT Greengrass Core software v2.4.0, v2.3.0, v2.2.0, and v2.1.0.
- Groups the test logs for each test case in a separate *<test-case-id>* folder within the *<device-tester-extract-location>*/results/*<execution-id>*/logs/*<test-group-id>* directory.
- Additional minor bug fixes.

Test suite version:

GGV2Q_2.0.1

- Released 2021.08.31

IDT v4.1.0 for AWS IoT Greengrass

Release notes:

- Enables you to validate and qualify devices running AWS IoT Greengrass Core software v2.3.0, v2.2.0, v2.1.0, and v2.0.5.
- Improves the `userdata.json` configuration by removing the requirement to specify the `GreengrassNucleusVersion` and `GreengrassCLIVersion` properties.
- Includes support for Lambda and MQTT feature qualification for AWS IoT Greengrass Core software v2.1.0 and later versions. You can now use IDT for AWS IoT Greengrass V2 to validate that your core device can run Lambda functions and that the device can publish and subscribe to AWS IoT Core MQTT topics.
- Improves logging capabilities.
- Additional minor bug fixes.

Test suite version:

GGV2Q_1.1.1

- Released 2021.06.18

IDT v4.0.2 for AWS IoT Greengrass

Release notes:

- Enables you to validate and qualify devices running AWS IoT Greengrass Core software v2.1.0.
- Adds support for Lambda and MQTT feature qualification for AWS IoT Greengrass Core software v2.1.0 and later versions. You can now use IDT for AWS IoT Greengrass V2 to validate that your core device can run Lambda functions and that the device can publish and subscribe to AWS IoT Core MQTT topics.

- Improves logging capabilities.
- Additional minor bug fixes.

Test suite version:

`GGV2Q_1.1.1`

- Released 2021.05.05

IDT v4.0.1 for AWS IoT Greengrass

Release notes:

- Enables you to validate and qualify devices running AWS IoT Greengrass Version 2 software.
- Enables you to develop and run your custom test suites using AWS IoT Device Tester for AWS IoT Greengrass. For more information, see Use IDT to develop and run your own test suites.
- Provides code signed IDT applications for macOS and Windows. On macOS, you might need to grant a security exception for IDT. For more information, see Security exception on macOS.

Test suite version:

`GGV2Q_1.0.0`

- Released 2020.12.22
- The test suite runs only required tests for qualification, unless you set the corresponding `value` in the `features` array to `yes`.

Download IDT for AWS IoT Greengrass V2

This topic describes the options to download AWS IoT Device Tester for AWS IoT Greengrass V2. You can either use one of the following software download links or you can follow instructions to programmatically download IDT.

Topics

- Download IDT manually
- Download IDT programmatically

By downloading the software, you agree to the AWS IoT Device Tester License Agreement.

> **Note**
>
> IDT does not support being run by multiple users from a shared location, such as an NFS directory or a Windows network shared folder. We recommend that you extract the IDT package to a local drive and run the IDT binary on your local workstation.

Download IDT manually

This topic lists supported versions of IDT for AWS IoT Greengrass V2. As a best practice, we recommend that you use the latest version of IDT for AWS IoT Greengrass V2 that supports your target version of AWS IoT Greengrass V2. New releases of AWS IoT Greengrass might require you to download a new version of IDT for AWS IoT Greengrass V2. You receive a notification when you start a test run if IDT for AWS IoT Greengrass V2 is not compatible with the version of AWS IoT Greengrass you are using.

IDT v4.9.4 for AWS IoT Greengrass

Supported AWS IoT Greengrass versions:

- Greengrass nucleus v2.12.0, v2.11.0, v2.10.0, and v2.9.5

IDT software downloads:

- IDT v4.9.4 with test suite GGV2Q_2.5.4 for Linux
- IDT v4.9.4 with test suite GGV2Q_2.5.4 for macOS
- IDT v4.9.4 with test suite GGV2Q_2.5.4 for Windows

Release notes:

- Enables device validation and qualification for devices running AWS IoT Greengrass Core software versions 2.12.0, 2.11.0, 2.10.0, and 2.9.5.
- Removes stream manager and machine learning test groups.

Additional notes:

- If your device uses a HSM and you are using nucleus 2.10.x, migrate to Greengrass nucleus version 2.11.0 or later.

Test suite version:

`GGV2Q_2.5.4`

- Released 2024.05.03

Download IDT programmatically

IDT provides an API operation that you can use to retrieve a URL where you can download IDT programmatically. You can also use this API operation to check if you have the latest version of IDT. This API operation has the following endpoint.

```
https://download.devicetester.iotdevicesecosystem.amazonaws.com/latestidt
```

To call this API operation, you must have permission to perform the **`iot-device-tester:LatestIdt`** action. Include your AWS signature and use `iot-device-tester` as the service name.

API request

HostOs – The operating system of the host machine. Choose from the following options:

- `mac`
- `linux`
- `windows`

TestSuiteType – The type of the test suite. Choose the following option:

GGV2 – IDT for AWS IoT Greengrass V2

ProductVersion

(Optional) The version of the Greengrass nucleus. The service returns the latest compatible version of IDT for that version of the Greengrass nucleus. If you don't specify this option, the service returns the latest version of IDT.

API response

The API response has the following format. The `DownloadURL` includes a zip file.

```
{
    "Success": True or False,
    "Message": Message,
    "LatestBk": {
        "Version": The version of the IDT binary,
        "TestSuiteVersion": The version of the test suite,
```

```
        "DownloadURL": The URL to download the IDT Bundle, valid for one hour
    }
}
```

Examples

You can reference the following examples to programmatically download IDT. These examples use credentials that you store in the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables. To follow best security practices, don't store your credentials in your code.

Example: Download using cURL version 7.75.0 or later (Mac and Linux)

If you have cURL version 7.75.0 or later, you can use the `aws-sigv4` flag to sign the API request. This example uses jq to parse the download URL from the response.

> **Warning**
>
> The `aws-sigv4` flag requires the query parameters of the curl GET request be in the order of **HostOs/ProductVersion/TestSuiteType** or **HostOs/TestSuiteType**. Orders that do not conform, will result in an error of getting mismatched signatures for the Canonical String from the API Gateway.
>
> If the optional parameter **ProductVersion** is included, you must use a supported product version as documented in Supported versions of AWS IoT Device Tester for AWS IoT Greengrass V2.

- Replace *us-west-2* with your AWS Region. For the list of Region codes, see Regional endpoints.
- Replace *linux* with your host machine's operating system.
- Replace *2.5.3* with your version of AWS IoT Greengrass nucleus.

```
url=$(curl --request GET "https://
download.devicetester.iotdevicesecosystem.amazonaws.com/latestidt?
HostOs=linux&ProductVersion=2.5.3&TestSuiteType=GGV2" \
--user $AWS_ACCESS_KEY_ID:$AWS_SECRET_ACCESS_KEY \
--aws-sigv4 "aws:amz:us-west-2:iot-device-tester" \
| jq -r '.LatestBk["DownloadURL"]')

curl $url --output devicetester.zip
```

Example: Download using an earlier version of cURL (Mac and Linux)

You can use the following cURL command with an AWS signature that you sign and calculate. For more information about how to sign and calculate an AWS signature, see Signing AWS API requests.

- Replace *linux* with your host machine's operating system.
- Replace *Timestamp* with the date and time, such as **20220210T004606Z**.
- Replace *Date* with the date, such as **20220210**.
- Replace *AWSRegion* with your AWS Region. For the list of Region codes, see Regional endpoints.
- Replace *AWSSignature* with the AWS signature that you generate.

```
curl --location --request GET 'https://
download.devicetester.iotdevicesecosystem.amazonaws.com/latestidt?
HostOs=linux&TestSuiteType=GGV2' \
--header 'X-Amz-Date: Timestamp \
--header 'Authorization: AWS4-HMAC-SHA256 Credential=$AWS_ACCESS_KEY_ID/Date/AWSRegion/
iot-device-tester/aws4_request, SignedHeaders=host;x-amz-date, Signature=AWSSignature'
```

Example: Download using a Python script

This example uses the Python requests library. This example is adapted from the Python example to Sign an AWS API request in the *AWS General Reference*.

- Replace *us-west-2* with your Region. For the list of Region codes, see Regional endpoints.
- Replace *linux* with your host machine's operating system.

```
# Copyright 2010-2022 Amazon.com, Inc. or its affiliates. All Rights Reserved.
#
# This file is licensed under the Apache License, Version 2.0 (the "License").
# You may not use this file except in compliance with the License. A copy of the
#License is located at
#
# http://aws.amazon.com/apache2.0/
#
# This file is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS
# OF ANY KIND, either express or implied. See the License for the specific
# language governing permissions and limitations under the License.
```

```
# See: http://docs.aws.amazon.com/general/latest/gr/sigv4_signing.html
# This version makes a GET request and passes the signature
# in the Authorization header.
import sys, os, base64, datetime, hashlib, hmac
import requests # pip install requests
# ************* REQUEST VALUES *************
method = 'GET'
service = 'iot-device-tester'
host = 'download.devicetester.iotdevicesecosystem.amazonaws.com'
region = 'us-west-2'
endpoint = 'https://download.devicetester.iotdevicesecosystem.amazonaws.com/latestidt'
request_parameters = 'HostOs=linux&TestSuiteType=GGV2'

# Key derivation functions. See:
# http://docs.aws.amazon.com/general/latest/gr/signature-v4-examples.html#signature-v4-
examples-python
def sign(key, msg):
    return hmac.new(key, msg.encode('utf-8'), hashlib.sha256).digest()

def getSignatureKey(key, dateStamp, regionName, serviceName):
    kDate = sign(('AWS4' + key).encode('utf-8'), dateStamp)
    kRegion = sign(kDate, regionName)
    kService = sign(kRegion, serviceName)
    kSigning = sign(kService, 'aws4_request')
    return kSigning

# Read AWS access key from env. variables or configuration file. Best practice is NOT
# to embed credentials in code.
access_key = os.environ.get('AWS_ACCESS_KEY_ID')
secret_key = os.environ.get('AWS_SECRET_ACCESS_KEY')
if access_key is None or secret_key is None:
    print('No access key is available.')
    sys.exit()

# Create a date for headers and the credential string
t = datetime.datetime.utcnow()
amzdate = t.strftime('%Y%m%dT%H%M%SZ')
datestamp = t.strftime('%Y%m%d') # Date w/o time, used in credential scope

# ************* TASK 1: CREATE A CANONICAL REQUEST *************
# http://docs.aws.amazon.com/general/latest/gr/sigv4-create-canonical-request.html
# Step 1 is to define the verb (GET, POST, etc.)--already done.
# Step 2: Create canonical URI--the part of the URI from domain to query
```

```
# string (use '/' if no path)
canonical_uri = '/latestidt'
# Step 3: Create the canonical query string. In this example (a GET request),
# request parameters are in the query string. Query string values must
# be URL-encoded (space=%20). The parameters must be sorted by name.
# For this example, the query string is pre-formatted in the request_parameters
 variable.
canonical_querystring = request_parameters
# Step 4: Create the canonical headers and signed headers. Header names
# must be trimmed and lowercase, and sorted in code point order from
# low to high. Note that there is a trailing \n.
canonical_headers = 'host:' + host + '\n' + 'x-amz-date:' + amzdate + '\n'
# Step 5: Create the list of signed headers. This lists the headers
# in the canonical_headers list, delimited with ";" and in alpha order.
# Note: The request can include any headers; canonical_headers and
# signed_headers lists those that you want to be included in the
# hash of the request. "Host" and "x-amz-date" are always required.
signed_headers = 'host;x-amz-date'
# Step 6: Create payload hash (hash of the request body content). For GET
# requests, the payload is an empty string ("").
payload_hash = hashlib.sha256(('').encode('utf-8')).hexdigest()
# Step 7: Combine elements to create canonical request
canonical_request = method + '\n' + canonical_uri + '\n' + canonical_querystring + '\n'
 + canonical_headers + '\n' + signed_headers + '\n' + payload_hash

# ************* TASK 2: CREATE THE STRING TO SIGN*************
# Match the algorithm to the hashing algorithm you use, either SHA-1 or
# SHA-256 (recommended)
algorithm = 'AWS4-HMAC-SHA256'
credential_scope = datestamp + '/' + region + '/' + service + '/' + 'aws4_request'
string_to_sign = algorithm + '\n' +  amzdate + '\n' +  credential_scope + '\n' +
 hashlib.sha256(canonical_request.encode('utf-8')).hexdigest()
# ************* TASK 3: CALCULATE THE SIGNATURE *************
# Create the signing key using the function defined above.
signing_key = getSignatureKey(secret_key, datestamp, region, service)
# Sign the string_to_sign using the signing_key
signature = hmac.new(signing_key, (string_to_sign).encode('utf-8'),
 hashlib.sha256).hexdigest()

# ************* TASK 4: ADD SIGNING INFORMATION TO THE REQUEST *************
# The signing information can be either in a query string value or in
# a header named Authorization. This code shows how to use a header.
# Create authorization header and add to request headers
```

```
authorization_header = algorithm + ' ' + 'Credential=' + access_key + '/' +
 credential_scope + ', ' +  'SignedHeaders=' + signed_headers + ', ' + 'Signature=' +
 signature
# The request can include any headers, but MUST include "host", "x-amz-date",
# and (for this scenario) "Authorization". "host" and "x-amz-date" must
# be included in the canonical_headers and signed_headers, as noted
# earlier. Order here is not significant.
# Python note: The 'host' header is added automatically by the Python 'requests'
 library.
headers = {'x-amz-date':amzdate, 'Authorization':authorization_header}

# ************* SEND THE REQUEST *************
request_url = endpoint + '?' + canonical_querystring
print('\nBEGIN REQUEST++++++++++++++++++++++++++++++++++++')
print('Request URL = ' + request_url)
response = requests.get(request_url, headers=headers)
print('\nRESPONSE++++++++++++++++++++++++++++++++++++')
print('Response code: %d\n' % response.status_code)
print(response.text)

download_url = response.json()["LatestBk"]["DownloadURL"]
r = requests.get(download_url)
open('devicetester.zip', 'wb').write(r.content)
```

Use IDT to run the AWS IoT Greengrass qualification suite

You can use AWS IoT Device Tester for AWS IoT Greengrass V2 to verify that the AWS IoT Greengrass Core software runs on your hardware and can communicate with the AWS Cloud. It also performs end-to-end tests with AWS IoT Core. For example, it verifies that your device can deploy components and upgrade them.

In addition to testing devices, IDT for AWS IoT Greengrass V2 creates resources (for example, AWS IoT things, groups, and so on) in your AWS account to facilitate the qualification process.

To create these resources, IDT for AWS IoT Greengrass V2 uses the AWS credentials configured in the `config.json` file to make API calls on your behalf. These resources are provisioned at various times during a test.

When you use IDT for AWS IoT Greengrass V2 to run the AWS IoT Greengrass qualification suite, it performs the following steps:

1. Loads and validates your device and credentials configuration.

2. Performs selected tests with the required local and cloud resources.
3. Cleans up local and cloud resources.
4. Generates tests reports that indicate if your board passed the tests required for qualification.

Test suite versions

IDT for AWS IoT Greengrass V2 organizes tests into test suites and test groups.

- A test suite is the set of test groups used to verify that a device works with particular versions of AWS IoT Greengrass.
- A test group is the set of individual tests related to a particular feature, such as component deployments.

Test suites are versioned using a *`major`*`.`*`minor`*`.`*`patch`* format, for example `GGV2Q_1.0.0`. When you download IDT, the package includes the latest Greengrass qualification suite version.

> **Important**
>
> Tests from unsupported test suite versions are not valid for device qualification. IDT doesn't print qualification reports for unsupported versions. For more information, see the section called “Support policy for AWS IoT Device Tester for AWS IoT Greengrass”.
> You can run `list-supported-products` to list the versions of AWS IoT Greengrass and test suites that are supported by your current version of IDT.

Test group descriptions

Required Test Groups for Core Qualification

These test groups are required to qualify your AWS IoT Greengrass V2 device for the AWS Partner Device Catalog.

Core Dependencies

Validates that the device meets all software and hardware requirements for the AWS IoT Greengrass Core software. This test group includes the following test case:

Java Version

Checks that the required Java version is installed on the device under test. AWS IoT Greengrass requires Java 8 or later.

PreTest Validation

Checks that the device meets the software requirements to run tests.

- For Linux-based devices, this test checks if the device can run the following Linux commands:

  `chmod`, `cp`, `echo`, `grep`, `kill`, `ln`, `mkinfo`, `ps`, `rm`, `sh`, `uname`
- For Windows-based devices, this test checks if the device has the following Microsoft software installed:

  Powershell v5.1 or later, .NET v4.6.1 or later, Visual C++ 2017 or later, PsExec utility

Version Checker

Checks that the version of AWS IoT Greengrass provided is compatible with the AWS IoT Device Tester version you are using.

Component

Validates that the device can deploy components and upgrade them. This test group includes the following tests:

Cloud Component

Validates device capability for cloud components.

Local Component

Validates device capability for local components.

Lambda

This test is not applicable for Windows-based devices.

Validates that the device can deploy Lambda function components that use the Java runtime, and that the Lambda functions can use AWS IoT Core MQTT topics as event sources for work messages.

MQTT

Validates that the device can subscribe and publish to AWS IoT Core MQTT topics.

Optional Test Groups

> **Note**
>
> These test groups are optional, and used only for qualifying Linux-based Greengrass core devices. If you choose to qualify for optional tests, your device is listed with additional capabilities in the AWS Partner Device Catalog.

Docker dependencies

Validates that the device meets all required technical dependencies to use the AWS-provided Docker application manager (`aws.greengrass.DockerApplicationManager`) component.

Docker Application Manager Qualification

Validates that the device can download a Docker container image from Amazon ECR .

Machine Learning Dependencies

> **Note**
>
> The machine learning optional test group is supported only in IDT v4.9.3.

Validates that the device meets all of the required technical dependencies to use the AWS-provided machine learning (ML) components.

Machine Learning Inference Tests

> **Note**
>
> The machine learning optional test group is supported only in IDT v4.9.3.

Validates that the device can perform ML inference using the Deep Learning Runtime and TensorFlow Lite ML frameworks.

Stream Manager Dependencies

> **Note**
>
> The stream manager optional test group is supported only in IDT v4.9.3.

Validates that the device can download, install, and run the AWS IoT Greengrass stream manager.

Hardware Security Integration (HSI)

> **Note**
>
> This test is available in IDT v4.9.3 and later for Linux-based devices only. AWS IoT Greengrass doesn't currently support hardware security integration for Windows devices.

Validates that the device can authenticate connections to the AWS IoT and AWS IoT Greengrass services using a private key and certificate that are stored in a hardware security module (HSM). This test also verifies that the AWS-provided PKCS#11 provider component can interface with the HSM using a vendor-provided PKCS#11 library. For more information, see Hardware security integration.

Prerequisites for running the AWS IoT Greengrass qualification suite

This section describes the prerequisites for using AWS IoT Device Tester (IDT) for AWS IoT Greengrass.

Download the latest version of AWS IoT Device Tester for AWS IoT Greengrass

Download the latest version of IDT and extract the software into a location (*<device-tester-extract-location>*) on your file system where you have read/write permissions.

> **Note**
>
> IDT does not support being run by multiple users from a shared location, such as an NFS directory or a Windows network shared folder. We recommend that you extract the IDT package to a local drive and run the IDT binary on your local workstation.

Windows has a path length limitation of 260 characters. If you are using Windows, extract IDT to a root directory like `C:\` or `D:\` to keep your paths under the 260 character limit.

Download the AWS IoT Greengrass software

IDT for AWS IoT Greengrass V2 tests your device for compatibility with a specific version of AWS IoT Greengrass. Run the following command to download the AWS IoT Greengrass Core software to a file named `aws.greengrass.nucleus.zip`. Replace *`version`* with a supported nucleus component version for your IDT version.

Linux or Unix

```
curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-version.zip >
 aws.greengrass.nucleus.zip
```

Windows Command Prompt (CMD)

```
curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-version.zip >
 aws.greengrass.nucleus.zip
```

PowerShell

```
iwr -Uri https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-version.zip -
OutFile aws.greengrass.nucleus.zip
```

Place the downloaded `aws.greengrass.nucleus.zip` file in the *`<device-tester-extract-location>`*`/products/` folder.

Note

Do not place multiple files in this directory for the same operating system and architecture.

Create and configure an AWS account

Before you can use AWS IoT Device Tester for AWS IoT Greengrass V2, you must perform the following steps:

1. Set up an AWS account. If you already have an AWS account, skip to step 2.
2. Configure permissions for IDT.

These account permissions allow IDT to access AWS services and create AWS resources, such as AWS IoT things and AWS IoT Greengrass components, on your behalf.

To create these resources, IDT for AWS IoT Greengrass V2 uses the AWS credentials configured in the `config.json` file to make API calls on your behalf. These resources are provisioned at various times during a test.

> **Note**
>
> Although most tests qualify for AWS Free Tier, you must provide a credit card when you sign up for an AWS account. For more information, see Why do I need a payment method if my account is covered by the Free Tier?.

Step 2: Configure permissions for IDT

In this step, configure the permissions that IDT for AWS IoT Greengrass V2 uses to run tests and collect IDT usage data. You can use the AWS Management Console or AWS Command Line Interface (AWS CLI) to create an IAM policy and a test user for IDT, and then attach policies to the user. If you already created a test user for IDT, skip to Configure your device to run IDT tests.

To configure permissions for IDT (console)

1. Sign in to the IAM console.
2. Create a customer managed policy that grants permissions to create roles with specific permissions.
   a. In the navigation pane, choose **Policies**, and then choose **Create policy**.
   b. If you are not using PreInstalled, on the **JSON** tab, replace the placeholder content with the following policy. If you are using PreInstalled, proceed to the following step.

```
{
    "Version":"2012-10-17",
    "Statement":[
    {
      "Sid":"passRoleForResources",
```

```
      "Effect":"Allow",
      "Action":"iam:PassRole",
      "Resource":"arn:aws:iam::*:role/idt-*",
      "Condition":{
        "StringEquals":{
          "iam:PassedToService":[
            "iot.amazonaws.com",
            "lambda.amazonaws.com",
            "greengrass.amazonaws.com"
          ]
        }
      }
    },
    {
      "Sid":"lambdaResources",
      "Effect":"Allow",
      "Action":[
        "lambda:CreateFunction",
        "lambda:PublishVersion",
        "lambda:DeleteFunction",
        "lambda:GetFunction"
      ],
      "Resource":[
        "arn:aws:lambda:*:*:function:idt-*"
      ]
    },
    {
      "Sid":"iotResources",
      "Effect":"Allow",
      "Action":[
        "iot:CreateThing",
        "iot:DeleteThing",
        "iot:DescribeThing",
        "iot:CreateThingGroup",
        "iot:DeleteThingGroup",
        "iot:DescribeThingGroup",
        "iot:AddThingToThingGroup",
        "iot:RemoveThingFromThingGroup",
        "iot:AttachThingPrincipal",
        "iot:DetachThingPrincipal",
        "iot:UpdateCertificate",
        "iot:DeleteCertificate",
        "iot:CreatePolicy",
        "iot:AttachPolicy",
```

```
        "iot:DetachPolicy",
        "iot:DeletePolicy",
        "iot:GetPolicy",
        "iot:Publish",
        "iot:TagResource",
        "iot:ListThingPrincipals",
        "iot:ListAttachedPolicies",
        "iot:ListTargetsForPolicy",
        "iot:ListThingGroupsForThing",
        "iot:ListThingsInThingGroup",
        "iot:CreateJob",
        "iot:DescribeJob",
        "iot:DescribeJobExecution",
        "iot:CancelJob"
      ],
      "Resource":[
        "arn:aws:iot:*:*:thing/idt-*",
        "arn:aws:iot:*:*:thinggroup/idt-*",
        "arn:aws:iot:*:*:policy/idt-*",
        "arn:aws:iot:*:*:cert/*",
        "arn:aws:iot:*:*:topic/idt-*",
        "arn:aws:iot:*:*:job/*"
      ]
    },
    {
      "Sid":"s3Resources",
      "Effect":"Allow",
      "Action":[
        "s3:GetObject",
        "s3:PutObject",
        "s3:DeleteObjectVersion",
        "s3:DeleteObject",
        "s3:CreateBucket",
        "s3:ListBucket",
        "s3:ListBucketVersions",
        "s3:DeleteBucket",
        "s3:PutObjectTagging",
        "s3:PutBucketTagging"
      ],
      "Resource":"arn:aws:s3::*:idt-*"
    },
    {
      "Sid":"roleAliasResources",
      "Effect":"Allow",
```

```
      "Action":[
        "iot:CreateRoleAlias",
        "iot:DescribeRoleAlias",
        "iot:DeleteRoleAlias",
        "iot:TagResource",
        "iam:GetRole"
      ],
      "Resource":[
        "arn:aws:iot:*:*:rolealias/idt-*",
        "arn:aws:iam::*:role/idt-*"
      ]
    },
    {
      "Sid":"idtExecuteAndCollectMetrics",
      "Effect":"Allow",
      "Action":[
        "iot-device-tester:SendMetrics",
        "iot-device-tester:SupportedVersion",
        "iot-device-tester:LatestIdt",
        "iot-device-tester:CheckVersion",
        "iot-device-tester:DownloadTestSuite"
      ],
      "Resource":"*"
    },
    {
      "Sid":"genericResources",
      "Effect":"Allow",
      "Action":[
        "greengrass:*",
        "iot:GetThingShadow",
        "iot:UpdateThingShadow",
        "iot:ListThings",
        "iot:DescribeEndpoint",
        "iot:CreateKeysAndCertificate"
      ],
      "Resource":"*"
    },
    {
      "Sid":"iamResourcesUpdate",
      "Effect":"Allow",
      "Action":[
        "iam:CreateRole",
        "iam:DeleteRole",
        "iam:CreatePolicy",
```

```
        "iam:DeletePolicy",
        "iam:AttachRolePolicy",
        "iam:DetachRolePolicy",
        "iam:TagRole",
        "iam:TagPolicy",
        "iam:GetPolicy",
        "iam:ListAttachedRolePolicies",
        "iam:ListEntitiesForPolicy"
      ],
      "Resource":[
        "arn:aws:iam::*:role/idt-*",
        "arn:aws:iam::*:policy/idt-*"
      ]
    }
  ]
}
```

c. If you are using PreInstalled, on the **JSON** tab, replace the placeholder content with the following policy. Make sure you:

- Replace *thingName* and *thingGroup* in the `iotResources` statement with the thing name and thing group that were created during the Greengrass installation on your device under test (DUT) to add permissions.
- Replace the *passRole* and *roleAlias* in the `roleAliasResources` statement and the `passRoleForResources` statement with the roles that were created during the Greengrass installation on your DUT.

```
{
    "Version":"2012-10-17",
    "Statement":[
    {
      "Sid":"passRoleForResources",
      "Effect":"Allow",
      "Action":"iam:PassRole",
      "Resource":"arn:aws:iam::*:role/passRole",
      "Condition":{
        "StringEquals":{
          "iam:PassedToService":[
            "iot.amazonaws.com",
            "lambda.amazonaws.com",
            "greengrass.amazonaws.com"
```

```
          ]
        }
      }
    },
    {
      "Sid":"lambdaResources",
      "Effect":"Allow",
      "Action":[
        "lambda:CreateFunction",
        "lambda:PublishVersion",
        "lambda:DeleteFunction",
        "lambda:GetFunction"
      ],
      "Resource":[
        "arn:aws:lambda:*:*:function:idt-*"
      ]
    },
    {
      "Sid":"iotResources",
      "Effect":"Allow",
      "Action":[
        "iot:CreateThing",
        "iot:DeleteThing",
        "iot:DescribeThing",
        "iot:CreateThingGroup",
        "iot:DeleteThingGroup",
        "iot:DescribeThingGroup",
        "iot:AddThingToThingGroup",
        "iot:RemoveThingFromThingGroup",
        "iot:AttachThingPrincipal",
        "iot:DetachThingPrincipal",
        "iot:UpdateCertificate",
        "iot:DeleteCertificate",
        "iot:CreatePolicy",
        "iot:AttachPolicy",
        "iot:DetachPolicy",
        "iot:DeletePolicy",
        "iot:GetPolicy",
        "iot:Publish",
        "iot:TagResource",
        "iot:ListThingPrincipals",
        "iot:ListAttachedPolicies",
        "iot:ListTargetsForPolicy",
        "iot:ListThingGroupsForThing",
```

```
        "iot:ListThingsInThingGroup",
        "iot:CreateJob",
        "iot:DescribeJob",
        "iot:DescribeJobExecution",
        "iot:CancelJob"
      ],
      "Resource":[
        "arn:aws:iot:*:*:thing/thingName",
        "arn:aws:iot:*:*:thinggroup/thingGroup",
        "arn:aws:iot:*:*:policy/idt-*",
        "arn:aws:iot:*:*:cert/*",
        "arn:aws:iot:*:*:topic/idt-*",
        "arn:aws:iot:*:*:job/*"
      ]
    },
    {
      "Sid":"s3Resources",
      "Effect":"Allow",
      "Action":[
        "s3:GetObject",
        "s3:PutObject",
        "s3:DeleteObjectVersion",
        "s3:DeleteObject",
        "s3:CreateBucket",
        "s3:ListBucket",
        "s3:ListBucketVersions",
        "s3:DeleteBucket",
        "s3:PutObjectTagging",
        "s3:PutBucketTagging"
      ],
      "Resource":"arn:aws:s3::*:idt-*"
    },
    {
      "Sid":"roleAliasResources",
      "Effect":"Allow",
      "Action":[
        "iot:CreateRoleAlias",
        "iot:DescribeRoleAlias",
        "iot:DeleteRoleAlias",
        "iot:TagResource",
        "iam:GetRole"
      ],
      "Resource":[
        "arn:aws:iot:*:*:rolealias/roleAlias",
```

```
        "arn:aws:iam::*:role/idt-*"
      ]
    },
    {
      "Sid":"idtExecuteAndCollectMetrics",
      "Effect":"Allow",
      "Action":[
        "iot-device-tester:SendMetrics",
        "iot-device-tester:SupportedVersion",
        "iot-device-tester:LatestIdt",
        "iot-device-tester:CheckVersion",
        "iot-device-tester:DownloadTestSuite"
      ],
      "Resource":"*"
    },
    {
      "Sid":"genericResources",
      "Effect":"Allow",
      "Action":[
        "greengrass:*",
        "iot:GetThingShadow",
        "iot:UpdateThingShadow",
        "iot:ListThings",
        "iot:DescribeEndpoint",
        "iot:CreateKeysAndCertificate"
      ],
      "Resource":"*"
    },
    {
      "Sid":"iamResourcesUpdate",
      "Effect":"Allow",
      "Action":[
        "iam:CreateRole",
        "iam:DeleteRole",
        "iam:CreatePolicy",
        "iam:DeletePolicy",
        "iam:AttachRolePolicy",
        "iam:DetachRolePolicy",
        "iam:TagRole",
        "iam:TagPolicy",
        "iam:GetPolicy",
        "iam:ListAttachedRolePolicies",
        "iam:ListEntitiesForPolicy"
      ],
```

```
      "Resource":[
        "arn:aws:iam::*:role/idt-*",
        "arn:aws:iam::*:policy/idt-*"
      ]
    }
  ]
}
```

> **Note**
>
> If you want to use a custom IAM role as the token exchange role for your device under test, make sure you update the `roleAliasResources` statement and the `passRoleForResources` statement in your policy to allow your custom IAM role resource.

d. Choose **Review policy**.

e. For **Name**, enter **`IDTGreengrassIAMPermissions`**. Under **Summary**, review the permissions granted by your policy.

f. Choose **Create policy**.

3. Create an IAM user and attach the permissions required by IDT for AWS IoT Greengrass.

   a. Create an IAM user. Follow steps 1 through 5 in Creating IAM users (console) in the *IAM User Guide*.

   b. Attach the permissions to your IAM user:

      i. On the **Set permissions** page, choose **Attach existing policies to user directly**.

      ii. Search for the **IDTGreengrassIAMPermissions** policy that you created in the previous step. Select the check box.

   c. Choose **Next: Tags**.

   d. Choose **Next: Review** to view a summary of your choices.

   e. Choose **Create user**.

   f. To view the user's access keys (access key IDs and secret access keys), choose **Show** next to the password and access key. To save the access keys, choose **Download.csv** and save the file to a secure location. You use this information later to configure your AWS credentials file.

4. Next step: Configure your physical device.

To configure permissions for IDT (AWS CLI)

1. On your computer, install and configure the AWS CLI if it's not already installed. Follow the steps in Installing the AWS CLI in the *AWS Command Line Interface User Guide*.

   > **Note**
   >
   > The AWS CLI is an open source tool that you can use to interact with AWS services from your command-line shell.

2. Create a customer managed policy that grants permissions to manage IDT and AWS IoT Greengrass roles.

   a. If you are not using PreInstalled, open a text editor and save the following policy contents in a JSON file. If you are using PreInstalled, proceed to the following step.

```
{
    "Version":"2012-10-17",
    "Statement":[
    {
      "Sid":"passRoleForResources",
      "Effect":"Allow",
      "Action":"iam:PassRole",
      "Resource":"arn:aws:iam::*:role/idt-*",
      "Condition":{
        "StringEquals":{
          "iam:PassedToService":[
            "iot.amazonaws.com",
            "lambda.amazonaws.com",
            "greengrass.amazonaws.com"
          ]
        }
      }
    },
    {
      "Sid":"lambdaResources",
      "Effect":"Allow",
      "Action":[
        "lambda:CreateFunction",
        "lambda:PublishVersion",
        "lambda:DeleteFunction",
        "lambda:GetFunction"
```

```
      ],
      "Resource":[
        "arn:aws:lambda:*:*:function:idt-*"
      ]
    },
    {
      "Sid":"iotResources",
      "Effect":"Allow",
      "Action":[
        "iot:CreateThing",
        "iot:DeleteThing",
        "iot:DescribeThing",
        "iot:CreateThingGroup",
        "iot:DeleteThingGroup",
        "iot:DescribeThingGroup",
        "iot:AddThingToThingGroup",
        "iot:RemoveThingFromThingGroup",
        "iot:AttachThingPrincipal",
        "iot:DetachThingPrincipal",
        "iot:UpdateCertificate",
        "iot:DeleteCertificate",
        "iot:CreatePolicy",
        "iot:AttachPolicy",
        "iot:DetachPolicy",
        "iot:DeletePolicy",
        "iot:GetPolicy",
        "iot:Publish",
        "iot:TagResource",
        "iot:ListThingPrincipals",
        "iot:ListAttachedPolicies",
        "iot:ListTargetsForPolicy",
        "iot:ListThingGroupsForThing",
        "iot:ListThingsInThingGroup",
        "iot:CreateJob",
        "iot:DescribeJob",
        "iot:DescribeJobExecution",
        "iot:CancelJob"
      ],
      "Resource":[
        "arn:aws:iot:*:*:thing/idt-*",
        "arn:aws:iot:*:*:thinggroup/idt-*",
        "arn:aws:iot:*:*:policy/idt-*",
        "arn:aws:iot:*:*:cert/*",
        "arn:aws:iot:*:*:topic/idt-*",
```

```
        "arn:aws:iot:*:*:job/*"
      ]
    },
    {
      "Sid":"s3Resources",
      "Effect":"Allow",
      "Action":[
        "s3:GetObject",
        "s3:PutObject",
        "s3:DeleteObjectVersion",
        "s3:DeleteObject",
        "s3:CreateBucket",
        "s3:ListBucket",
        "s3:ListBucketVersions",
        "s3:DeleteBucket",
        "s3:PutObjectTagging",
        "s3:PutBucketTagging"
      ],
      "Resource":"arn:aws:s3::*:idt-*"
    },
    {
      "Sid":"roleAliasResources",
      "Effect":"Allow",
      "Action":[
        "iot:CreateRoleAlias",
        "iot:DescribeRoleAlias",
        "iot:DeleteRoleAlias",
        "iot:TagResource",
        "iam:GetRole"
      ],
      "Resource":[
        "arn:aws:iot:*:*:rolealias/idt-*",
        "arn:aws:iam::*:role/idt-*"
      ]
    },
    {
      "Sid":"idtExecuteAndCollectMetrics",
      "Effect":"Allow",
      "Action":[
        "iot-device-tester:SendMetrics",
        "iot-device-tester:SupportedVersion",
        "iot-device-tester:LatestIdt",
        "iot-device-tester:CheckVersion",
        "iot-device-tester:DownloadTestSuite"
```

```
      ],
      "Resource":"*"
    },
    {
      "Sid":"genericResources",
      "Effect":"Allow",
      "Action":[
        "greengrass:*",
        "iot:GetThingShadow",
        "iot:UpdateThingShadow",
        "iot:ListThings",
        "iot:DescribeEndpoint",
        "iot:CreateKeysAndCertificate"
      ],
      "Resource":"*"
    },
    {
      "Sid":"iamResourcesUpdate",
      "Effect":"Allow",
      "Action":[
        "iam:CreateRole",
        "iam:DeleteRole",
        "iam:CreatePolicy",
        "iam:DeletePolicy",
        "iam:AttachRolePolicy",
        "iam:DetachRolePolicy",
        "iam:TagRole",
        "iam:TagPolicy",
        "iam:GetPolicy",
        "iam:ListAttachedRolePolicies",
        "iam:ListEntitiesForPolicy"
      ],
      "Resource":[
        "arn:aws:iam::*:role/idt-*",
        "arn:aws:iam::*:policy/idt-*"
      ]
    }
  ]
}
```

b. If you are using PreInstalled, open a text editor and save the following policy contents in a JSON file. Make sure you:

- Replace *thingName* and *thingGroup* in the `iotResources` statement that were created during the Greengrass installation on your device under test (DUT) to add permissions.
- Replace the *passRole* and *roleAlias* in the `roleAliasResources` statement and the `passRoleForResources` statement with the roles that were created during the Greengrass installation on your DUT.

```
{
    "Version":"2012-10-17",
    "Statement":[
    {
      "Sid":"passRoleForResources",
      "Effect":"Allow",
      "Action":"iam:PassRole",
      "Resource":"arn:aws:iam::*:role/passRole",
      "Condition":{
        "StringEquals":{
          "iam:PassedToService":[
            "iot.amazonaws.com",
            "lambda.amazonaws.com",
            "greengrass.amazonaws.com"
          ]
        }
      }
    },
    {
      "Sid":"lambdaResources",
      "Effect":"Allow",
      "Action":[
        "lambda:CreateFunction",
        "lambda:PublishVersion",
        "lambda:DeleteFunction",
        "lambda:GetFunction"
      ],
      "Resource":[
        "arn:aws:lambda:*:*:function:idt-*"
      ]
    },
    {
      "Sid":"iotResources",
```

```
      "Effect":"Allow",
      "Action":[
        "iot:CreateThing",
        "iot:DeleteThing",
        "iot:DescribeThing",
        "iot:CreateThingGroup",
        "iot:DeleteThingGroup",
        "iot:DescribeThingGroup",
        "iot:AddThingToThingGroup",
        "iot:RemoveThingFromThingGroup",
        "iot:AttachThingPrincipal",
        "iot:DetachThingPrincipal",
        "iot:UpdateCertificate",
        "iot:DeleteCertificate",
        "iot:CreatePolicy",
        "iot:AttachPolicy",
        "iot:DetachPolicy",
        "iot:DeletePolicy",
        "iot:GetPolicy",
        "iot:Publish",
        "iot:TagResource",
        "iot:ListThingPrincipals",
        "iot:ListAttachedPolicies",
        "iot:ListTargetsForPolicy",
        "iot:ListThingGroupsForThing",
        "iot:ListThingsInThingGroup",
        "iot:CreateJob",
        "iot:DescribeJob",
        "iot:DescribeJobExecution",
        "iot:CancelJob"
      ],
      "Resource":[
        "arn:aws:iot:*:*:thing/thingName",
        "arn:aws:iot:*:*:thinggroup/thingGroup",
        "arn:aws:iot:*:*:policy/idt-*",
        "arn:aws:iot:*:*:cert/*",
        "arn:aws:iot:*:*:topic/idt-*",
        "arn:aws:iot:*:*:job/*"
      ]
    },
    {
      "Sid":"s3Resources",
      "Effect":"Allow",
      "Action":[
```

```
        "s3:GetObject",
        "s3:PutObject",
        "s3:DeleteObjectVersion",
        "s3:DeleteObject",
        "s3:CreateBucket",
        "s3:ListBucket",
        "s3:ListBucketVersions",
        "s3:DeleteBucket",
        "s3:PutObjectTagging",
        "s3:PutBucketTagging"
      ],
      "Resource":"arn:aws:s3::*:idt-*"
    },
    {
      "Sid":"roleAliasResources",
      "Effect":"Allow",
      "Action":[
        "iot:CreateRoleAlias",
        "iot:DescribeRoleAlias",
        "iot:DeleteRoleAlias",
        "iot:TagResource",
        "iam:GetRole"
      ],
      "Resource":[
        "arn:aws:iot:*:*:rolealias/roleAlias",
        "arn:aws:iam::*:role/idt-*"
      ]
    },
    {
      "Sid":"idtExecuteAndCollectMetrics",
      "Effect":"Allow",
      "Action":[
        "iot-device-tester:SendMetrics",
        "iot-device-tester:SupportedVersion",
        "iot-device-tester:LatestIdt",
        "iot-device-tester:CheckVersion",
        "iot-device-tester:DownloadTestSuite"
      ],
      "Resource":"*"
    },
    {
      "Sid":"genericResources",
      "Effect":"Allow",
      "Action":[
```

```
        "greengrass:*",
        "iot:GetThingShadow",
        "iot:UpdateThingShadow",
        "iot:ListThings",
        "iot:DescribeEndpoint",
        "iot:CreateKeysAndCertificate"
      ],
      "Resource":"*"
    },
    {
      "Sid":"iamResourcesUpdate",
      "Effect":"Allow",
      "Action":[
        "iam:CreateRole",
        "iam:DeleteRole",
        "iam:CreatePolicy",
        "iam:DeletePolicy",
        "iam:AttachRolePolicy",
        "iam:DetachRolePolicy",
        "iam:TagRole",
        "iam:TagPolicy",
        "iam:GetPolicy",
        "iam:ListAttachedRolePolicies",
        "iam:ListEntitiesForPolicy"
      ],
      "Resource":[
        "arn:aws:iam::*:role/idt-*",
        "arn:aws:iam::*:policy/idt-*"
      ]
    }
  ]
}
```

Note

If you want to use a custom IAM role as the token exchange role for your device under test, make sure you update the `roleAliasResources` statement and the `passRoleForResources` statement in your policy to allow your custom IAM role resource.

c. Run the following command to create a customer managed policy named `IDTGreengrassIAMPermissions`. Replace *`policy.json`* with the full path to the JSON file that you created in the previous step.

```
aws iam create-policy --policy-name IDTGreengrassIAMPermissions --policy-
document file://policy.json
```

3. Create an IAM user and attach the permissions required by IDT for AWS IoT Greengrass.

a. Create an IAM user. In this example setup, the user is named `IDTGreengrassUser`.

```
aws iam create-user --user-name IDTGreengrassUser
```

b. Attach the `IDTGreengrassIAMPermissions` policy you created in step 2 to your IAM user. Replace *`<account-id>`* in the command with the ID of your AWS account.

```
aws iam attach-user-policy --user-name IDTGreengrassUser --policy-arn
 arn:aws:iam::<account-id>:policy/IDTGreengrassIAMPermissions
```

4. Create a secret access key for the user.

```
aws iam create-access-key --user-name IDTGreengrassUser
```

Store the output in a secure location. You use this information later to configure your AWS credentials file.

5. Next step: Configure your physical device.

AWS IoT Device Tester permissions

The following policies describe AWS IoT Device Tester permissions.

AWS IoT Device Tester requires these permissions for version checking and auto-update features.

- `iot-device-tester:SupportedVersion`

  Grants AWS IoT Device Tester permission to fetch the list of supported products, test suites and IDT versions.
- `iot-device-tester:LatestIdt`

  Grants AWS IoT Device Tester permission to fetch the latest IDT version available for download.

- `iot-device-tester:CheckVersion`

  Grants AWS IoT Device Tester permission to check version compatibility for IDT, test suites and products.

- `iot-device-tester:DownloadTestSuite`

  Grants AWS IoT Device Tester permission to download test suites updates.

AWS IoT Device Tester also uses the following permission for optional metrics reporting:

- `iot-device-tester:SendMetrics`

  Grants permission to AWS to collect metrics about AWS IoT Device Tester internal usage. If this permission is omitted, these metrics will not be collected.

Configure your device to run IDT tests

To enable IDT to run tests for device qualification, you must configure your host computer to access your device, and configure user permissions on your device.

Install Java on the host computer

Starting with IDT v4.2.0, the optional qualification tests for AWS IoT Greengrass require Java to run.

You can use Java version 8 or greater. We recommend that you use Amazon Corretto or OpenJDK long-term support versions. Version 8 or higher is required..

Configure your host computer to access your device under test

IDT runs on your host computer and must be able to use SSH to connect to your device. There are two options to allow IDT to gain SSH access to your devices under test:

1. Follow the instructions here to create an SSH key pair and authorize your key to sign in to your device under test without specifying a password.
2. Provide a user name and password for each device in the `device.json` file. For more information, see Configure device.json.

You can use any SSL implementation to create an SSH key. The following instructions show you how to use SSH-KEYGEN or PuTTYgen (for Windows). If you are using another SSL implementation, refer to the documentation for that implementation.

IDT uses SSH keys to authenticate with your device under test.

To create an SSH key with SSH-KEYGEN

1. Create an SSH key.

   You can use the Open SSH **ssh-keygen** command to create an SSH key pair. If you already have an SSH key pair on your host computer, it is a best practice to create a SSH key pair specifically for IDT. This way, after you have completed testing, your host computer can no longer connect to your device without entering a password. It also allows you to restrict access to the remote device to only those who need it.

   > **Note**
   >
   > Windows does not have an installed SSH client. For information about installing an SSH client on Windows, see Download SSH Client Software.

   The **ssh-keygen** command prompts you for a name and path to store the key pair. By default, the key pair files are named `id_rsa` (private key) and `id_rsa.pub` (public key). On macOS and Linux, the default location of these files is `~/.ssh/`. On Windows, the default location is `C:\Users\<user-name>\.ssh`.

   When prompted, enter a key phrase to protect your SSH key. For more information, see Generate a New SSH key.

2. Add authorized SSH keys to your device under test.

   IDT must use your SSH private key to sign in to your device under test. To authorize your SSH private key to sign in to your device under test, use the **ssh-copy-id** command from your host computer. This command adds your public key into the `~/.ssh/authorized_keys` file on your device under test. For example:

   $ ssh-copy-id *<remote-ssh-user>*@*<remote-device-ip>*

Where *remote-ssh-user* is the user name used to sign in to your device under test and *remote-device-ip* is the IP address of the device under test to run tests against. For example:

ssh-copy-id pi@192.168.1.5

When prompted, enter the password for the user name you specified in the **ssh-copy-id** command.

ssh-copy-id assumes the public key is named `id_rsa.pub` and is stored the default location (on macOS and Linux, `~/.ssh/` and on Windows, `C:\Users\`*<user-name>*`\.ssh`). If you gave the public key a different name or stored it in a different location, you must specify the fully qualified path to your SSH public key using the **-i** option to **ssh-copy-id** (for example, **ssh-copy-id -i ~/my/path/myKey.pub**). For more information about creating SSH keys and copying public keys, see SSH-COPY-ID.

To create an SSH key using PuTTYgen (Windows only)

1. Make sure you have the OpenSSH server and client installed on your device under test. For more information, see OpenSSH.
2. Install PuTTYgen on your device under test.
3. Open PuTTYgen.
4. Choose **Generate** and move your mouse cursor inside the box to generate a private key.
5. From the **Conversions** menu, choose **Export OpenSSH key**, and save the private key with a `.pem` file extension.
6. Add the public key to the `/home/`*<user>*`/.ssh/authorized_keys` file on device under test.
   a. Copy the public key text from the PuTTYgen window.
   b. Use PuTTY to create a session on your device under test.
      i. From a command prompt or Windows Powershell window, run the following command:

         C:/*<path-to-putty>*/putty.exe -ssh *<user>*@*<dut-ip-address>*

      ii. When prompted, enter your device's password.

iii. Use vi or another text editor to append the public key to the `/home/<user>/.ssh/authorized_keys` file on your device under test.

7. Update your `device.json` file with your user name, the IP address, and path to the private key file that you just saved on your host computer for each device under test. For more information, see the section called "Configure device.json". Make sure you provide the full path and file name to the private key and use forward slashes ('/'). For example, for the Windows path `C:\DT\privatekey.pem`, use `C:/DT/privatekey.pem` in the `device.json` file.

Configure user credentials for Windows devices

To qualify a Windows-based device, you must configure user credentials in the LocalSystem account on the device under test for the following users:

- The default Greengrass user (`ggc_user`).
- The user that you use to connect to the device under test. You configure this user in the `device.json` file.

You must create each user in the LocalSystem account on the device under test, and then store the user name and password for the user in the Credential Manager instance for the LocalSystem account.

To configure users on Windows devices

1. Open the Windows Command Prompt (`cmd.exe`) as an administrator.
2. Create the users in the LocalSystem account on the Windows device. Run the following command for each user that you want to create. For the default Greengrass user, replace *`user-name`* with `ggc_user`. Replace *`password`* with a secure password.

   ```
   net user /add user-name password
   ```

3. Download and install the PsExec utility from Microsoft on the device.
4. Use the PsExec utility to store the user name and password for the default user in the Credential Manager instance for the LocalSystem account.

   Run the following command for each user that you want to configure in Credential Manager. For the default Greengrass user, replace *`user-name`* with `ggc_user`. Replace *`password`* with the user's password that you set earlier.

```
psexec -s cmd /c cmdkey /generic:user-name /user:user-name /pass:password
```

If the **PsExec License Agreement** opens, choose **Accept** to agree to the license and run the command.

> **Note**
>
> On Windows devices, the LocalSystem account runs the Greengrass nucleus, and you must use the PsExec utility to store user information in the LocalSystem account. Using the Credential Manager application stores this information in the Windows account of the currently logged on user, instead of the LocalSystem account.

Configure user permissions on your device

IDT performs operations on various directories and files in a device under test. Some of these operations require elevated permissions (using **sudo**). To automate these operations, IDT for AWS IoT Greengrass V2 must be able to run commands with sudo without being prompted for a password.

Follow these steps on the device under test to allow sudo access without being prompted for a password.

> **Note**
>
> `username` refers to the SSH user used by IDT to access the device under test.

To add the user to the sudo group

1. On the device under test, run `sudo usermod -aG sudo <username>`.
2. Sign out and then sign back in for changes to take effect.
3. To verify your user name was added successfully, run **sudo echo test**. If you are not prompted for a password, your user is configured correctly.
4. Open the `/etc/sudoers` file and add the following line to the end of the file:

   ```
   <ssh-username> ALL=(ALL) NOPASSWD: ALL
   ```

Configure a custom token exchange role

You can choose to use a custom IAM role as the token exchange role that the device under test assumes to interact with AWS resources. For information about creating an IAM role, see Creating IAM roles in the *IAM User Guide*.

You must meet the following requirements to allow IDT to use your custom IAM role. We strongly recommend that you add only the minimum required policy actions to this role.

- The userdata.json configuration file must be updated to set the `GreengrassV2TokenExchangeRole` parameter to `true`.
- The custom IAM role must be configured with the following minimum trust policy:

  JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Effect":"Allow",
         "Principal":{
            "Service":[
               "credentials.iot.amazonaws.com",
               "lambda.amazonaws.com",
               "sagemaker.amazonaws.com"
            ]
         },
         "Action":"sts:AssumeRole"
      }
   ]
}
```

- The custom IAM role must be configured with the following minimum permissions policy:

  JSON

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Effect":"Allow",
         "Action":[
            "iot:DescribeCertificate",
```

```
            "logs:CreateLogGroup",
            "logs:CreateLogStream",
            "logs:PutLogEvents",
            "logs:DescribeLogStreams",
            "iot:Connect",
            "iot:Publish",
            "iot:Subscribe",
            "iot:Receive",
            "iot:ListThingPrincipals",
            "iot:GetThingShadow",
            "iot:UpdateThingShadow",
            "s3:GetBucketLocation",
            "s3:GetObject",
            "s3:PutObject",
            "s3:AbortMultipartUpload",
            "s3:ListMultipartUploadParts"
         ],
         "Resource":"*"
      }
   ]
}
```

- The name of the custom IAM role must match the IAM role resource that you specify in the IAM permissions for the test user. By default, the test user policy allows access to IAM roles that have the `idt-` prefix in their role names. If your IAM role name doesn't use this prefix, add the `arn:aws:iam::*:role/`*`custom-iam-role-name`* resource to the `roleAliasResources` statement and the `passRoleForResources` statement in your test user policy, as shown in the following examples:

  Example `passRoleForResources` statement

```
{
   "Sid":"passRoleForResources",
   "Effect":"Allow",
   "Action":"iam:PassRole",
   "Resource":"arn:aws:iam::*:role/custom-iam-role-name",
   "Condition":{
      "StringEquals":{
         "iam:PassedToService":[
            "iot.amazonaws.com",
            "lambda.amazonaws.com",
            "greengrass.amazonaws.com"
```

```
            ]
        }
    }
}
```

Example `roleAliasResources` statement

```
{
    "Sid":"roleAliasResources",
    "Effect":"Allow",
    "Action":[
        "iot:CreateRoleAlias",
        "iot:DescribeRoleAlias",
        "iot:DeleteRoleAlias",
        "iot:TagResource",
        "iam:GetRole"
    ],
    "Resource":[
        "arn:aws:iot:*:*:rolealias/idt-*",
        "arn:aws:iam::*:role/custom-iam-role-name"
    ]
}
```

Configure your device to test optional features

This section describes the device requirements to run IDT tests for optional Docker and machine learning (ML) features. The ML features are supported only in IDT v4.9.3. You must make sure your device meets these requirements only if you want to test these features. Otherwise, continue to the section called “Configure IDT settings”.

Topics

- Docker qualification requirements
- ML qualification requirements
- HSM qualification requirements

Docker qualification requirements

IDT for AWS IoT Greengrass V2 provides Docker qualification tests to validate that your devices can use the AWS-provided Docker application manager component to download Docker container

images that you can run using custom Docker container components. For information about creating custom Docker components, see Run a Docker container.

To run Docker qualification tests, your devices under test must meet the following requirements to deploy the Docker application manager component.

- Docker Engine 1.9.1 or later installed on the Greengrass core device. Versions up to and including 29.6.0, such as 20.10, 24.0, 25.0, and 27.5, work with the AWS IoT Greengrass Core software. You must install Docker directly on the core device before you deploy components that run Docker containers.
- The Docker daemon started and running on the core device before you deploy this component.
- The system user that runs a Docker container component must have root or administrator permissions, or you must configure Docker to run it as a non-root or non-admistrator user.
  - On Linux devices, you can add a user to the `docker` group to call `docker` commands without `sudo`.
  - On Windows devices, you can add a user to the `docker-users` group to call `docker` commands without adminstrator privileges.

  Linux or Unix

  To add `ggc_user`, or the non-root user that you use to run Docker container components, to the `docker` group, run the following command.

  ```
  sudo usermod -aG docker ggc_user
  ```

  For more information, see Manage Docker as a non-root user.

  Windows Command Prompt (CMD)

  To add `ggc_user`, or the user that you use to run Docker container components, to the `docker-users` group, run the following command as an administrator.

  ```
  net localgroup docker-users ggc_user /add
  ```

  Windows PowerShell

  To add `ggc_user`, or the user that you use to run Docker container components, to the `docker-users` group, run the following command as an administrator.

```
Add-LocalGroupMember -Group docker-users -Member ggc_user
```

ML qualification requirements

> **Note**
>
> The machine learning feature is supported only in IDT v4.9.3.

IDT for AWS IoT Greengrass V2 provides ML qualification tests to validate that your devices can use the AWS-provided machine learning components to perform ML inference locally using the Deep Learning Runtime or TensorFlow Lite ML frameworks. For more information about running ML inference on Greengrass devices, see Perform machine learning inference.

To run ML qualification tests, your devices under test must meet the following requirements to deploy the machine learning components.

- On Greengrass core devices running Amazon Linux 2 or Ubuntu 18.04, GNU C Library (glibc) version 2.27 or later installed on the device.
- On Armv7l devices, such as Raspberry Pi, dependencies for OpenCV-Python installed on the device. Run the following command to install the dependencies.

  ```
  sudo apt-get install libopenjp2-7 libilmbase23 libopenexr-dev libavcodec-dev
   libavformat-dev libswscale-dev libv4l-dev libgtk-3-0 libwebp-dev
  ```
- Raspberry Pi devices that run Raspberry Pi OS Bullseye must meet the following requirements:
  - NumPy 1.22.4 or later installed on the device. Raspberry Pi OS Bullseye includes an earlier version of NumPy, so you can run the following command to upgrade NumPy on the device.

    ```
    pip3 install --upgrade numpy
    ```
  - The legacy camera stack enabled on the device. Raspberry Pi OS Bullseye includes a new camera stack that is enabled by default and isn't compatible, so you must enable the legacy camera stack.

    To enable the legacy camera stack

    1. Run the following command to open the Raspberry Pi configuration tool.

```
sudo raspi-config
```

2. Select **Interface Options**.
3. Select **Legacy camera** to enable the legacy camera stack.
4. Reboot the Raspberry Pi.

HSM qualification requirements

AWS IoT Greengrass provides PKCS#11 provider component to integrate with the PKCS Hardware Security Module (HSM) on the device. The HSM setup depends on your device and the HSM module that you have chosen. As long as the expected HSM configuration, as documented in the IDT configuration settings, is provided, IDT will have the information required to run this optional feature qualification test.

Configure IDT settings to run the AWS IoT Greengrass qualification suite

Before you run tests, you must configure settings for AWS credentials and devices on your host computer.

Configure AWS credentials in config.json

You must configure your IAM user credentials in the *<device_tester_extract_location>*/`configs/config.json` file. Use the credentials for the IDT for AWS IoT Greengrass V2 user created in the section called "Create and configure an AWS account". You can specify your credentials in one of two ways:

- In a credentials file
- As environment variables

Configure AWS credentials with a credentials file

IDT uses the same credentials file as the AWS CLI. For more information, see Configuration and credential files.

The location of the credentials file varies, depending on the operating system you are using:

- macOS, Linux: `~/.aws/credentials`

- Windows: `C:\Users\UserName\.aws\credentials`

Add your AWS credentials to the `credentials` file in the following format:

```
[default]
aws_access_key_id = <your_access_key_id>
aws_secret_access_key = <your_secret_access_key>
```

To configure IDT for AWS IoT Greengrass V2 to use AWS credentials from your `credentials` file, edit your `config.json` file as follows:

```
{
  "awsRegion": "region",
  "auth": {
    "method": "file",
    "credentials": {
      "profile": "default"
    }
  }
}
```

> **Note**
>
> If you do not use the `default` AWS profile, be sure to change the profile name in your `config.json` file. For more information, see Named profiles.

Configure AWS credentials with environment variables

Environment variables are variables maintained by the operating system and used by system commands. They are not saved if you close the SSH session. IDT for AWS IoT Greengrass V2 can use the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables to store your AWS credentials.

To set these variables on Linux, macOS, or Unix, use **export**:

```
export AWS_ACCESS_KEY_ID=<your_access_key_id>
export AWS_SECRET_ACCESS_KEY=<your_secret_access_key>
```

To set these variables on Windows, use **set**:

```
set AWS_ACCESS_KEY_ID=<your_access_key_id>
set AWS_SECRET_ACCESS_KEY=<your_secret_access_key>
```

To configure IDT to use the environment variables, edit the `auth` section in your `config.json` file. Here is an example:

```
{
  "awsRegion": "region",
  "auth": {
    "method": "environment"
  }
}
```

Configure device.json

> **Note**
>
> IDT v4.9.3 supports testing the `ml`, `docker`, and `streamManagement` features. IDT v4.9.4 and later versions support testing `docker`. If you don't want to test these features, set the corresponding value to no.

In addition to AWS credentials, IDT for AWS IoT Greengrass V2 needs information about the devices that tests are run on. Example information would be IP address, login information, operating system, and CPU architecture.

You must provide this information using the `device.json` template located in *<device_tester_extract_location>*`/configs/device.json`:

IDT v4.9.3

```
[
  {
    "id": "<pool-id>",
    "sku": "<sku>",
    "features": [
      {
        "name": "arch",
        "value": "x86_64 | armv6l | armv7l | aarch64"
      },
```

```
      {
        "name": "ml",
        "value": "dlr | tensorflowlite | dlr,tensorflowlite | no"
      },
      {
        "name": "docker",
        "value": "yes | no"
      },
      {
        "name": "streamManagement",
        "value": "yes | no"
      },
      {
        "name": "hsi",
        "value": "hsm | no"
      }
    ],
    "devices": [
      {
        "id": "<device-id>",
        "operatingSystem": "Linux | Windows",
        "connectivity": {
          "protocol": "ssh",
          "ip": "<ip-address>",
          "port": 22,
          "publicKeyPath": "<public-key-path>",
          "auth": {
            "method": "pki | password",
            "credentials": {
              "user": "<user-name>",
              "privKeyPath": "/path/to/private/key",
              "password": "<password>"
            }
          }
        }
      }
    ]
  }
]
```

Note

Specify `privKeyPath` only if `method` is set to `pki`.

Specify `password` only if `method` is set to `password`.

All properties that contain values are required, as described here:

`id`

A user-defined alphanumeric ID that uniquely identifies a collection of devices called a *device pool*. Devices that belong to a pool must have identical hardware. When you run a suite of tests, devices in the pool are used to parallelize the workload. Multiple devices are used to run different tests.

`sku`

An alphanumeric value that uniquely identifies the device under test. The SKU is used to track qualified boards.

Note

If you want to list your device in the AWS Partner Device Catalog, the SKU you specify here must match the SKU that you use in the listing process.

`features`

An array that contains the device's supported features. All features are required.

`arch`

The supported operating system architectures that the test run validates. Valid values are:

- `x86_64`
- `armv6l`
- `armv7l`
- `aarch64`

`ml`

Validates that the device meets all of the required technical dependencies to use the AWS-provided machine learning (ML) components.

Enabling this feature also validates that the device can perform ML inference using the Deep Learning Runtime and TensorFlow Lite ML frameworks. .

Valid values are any combination of `dlr` and `tensorflowlite`, or no.

`docker`

Validates that the device meets all required technical dependencies to use the AWS-provided Docker application manager (`aws.greengrass.DockerApplicationManager`) component.

Enabling this feature also validates that the device can download a Docker container image from Amazon ECR . .

Valid values are any combination of `yes` or no.

`streamManagement`

Validates that the device can download, install, and run the AWS IoT Greengrass stream manager.

Valid values are any combination of `yes` or no.

`hsi`

Validates that the device can authenticate connections to the AWS IoT and AWS IoT Greengrass services using a private key and certificate that are stored in a hardware security module (HSM). This test also verifies that the AWS-provided PKCS#11 provider component can interface with the HSM using a vendor-provided PKCS#11 library. For more information, see Hardware security integration.

Valid values are `hsm` or no.

> **Note**
>
> Testing the `hsi` is available only with IDT v4.9.3 and later versions.

`devices.id`

A user-defined unique identifier for the device being tested.

`devices.operatingSystem`

The device operating system. Supported values are `Linux` and `Windows`.

`connectivity.protocol`

The communication protocol used to communicate with this device. Currently, the only supported value is `ssh` for physical devices.

`connectivity.ip`

The IP address of the device being tested.

This property applies only if `connectivity.protocol` is set to `ssh`.

`connectivity.port`

Optional. The port number to use for SSH connections.

The default value is 22.

This property applies only if `connectivity.protocol` is set to `ssh`.

`connectivity.publicKeyPath`

Optional. The full path to the public key used to authenticate connections to the device under test.

When you specify the `publicKeyPath`, IDT validates the device's public key when it establishes an SSH connection to the device under test. If this value is not specified, IDT creates an SSH connection, but doesn't validate the device's public key.

We strongly recommend that you specify the path to the public key, and that you use a secure method to fetch this public key. For standard command line-based SSH clients, the public key is provided in the `known_hosts` file. If you specify a separate public key file, this file must use the same format as the `known_hosts` file, that is, *`ip-address key-type public-key`*. If there are multiple entries with the same ip-address, the entry for the key-type used by IDT must be before the other entries in the file.

`connectivity.auth`

Authentication information for the connection.

This property applies only if `connectivity.protocol` is set to `ssh`.

`connectivity.auth.method`

The authentication method used to access a device over the given connectivity protocol.

Supported values are:

- `pki`
- `password`

`connectivity.auth.credentials`

The credentials used for authentication.

`connectivity.auth.credentials.password`

The password used for signing in to the device being tested.

This value applies only if `connectivity.auth.method` is set to `password`.

`connectivity.auth.credentials.privKeyPath`

The full path to the private key used to sign in to the device under test.

This value applies only if `connectivity.auth.method` is set to `pki`.

`connectivity.auth.credentials.user`

The user name for signing in to the device being tested.

IDT v4.9.4

```
[
  {
    "id": "<pool-id>",
    "sku": "<sku>",
    "features": [
      {
        "name": "arch",
        "value": "x86_64 | armv6l | armv7l | aarch64"
      },
      {
        "name": "docker",
        "value": "yes | no"
      },
```

```
      {
        "name": "hsi",
        "value": "hsm | no"
      }
    ],
    "devices": [
      {
        "id": "<device-id>",
        "operatingSystem": "Linux | Windows",
        "connectivity": {
          "protocol": "ssh",
          "ip": "<ip-address>",
          "port": 22,
          "publicKeyPath": "<public-key-path>",
          "auth": {
            "method": "pki | password",
            "credentials": {
              "user": "<user-name>",
              "privKeyPath": "/path/to/private/key",
              "password": "<password>"
            }
          }
        }
      }
    ]
  }
]
```

> **Note**
>
> Specify `privKeyPath` only if `method` is set to `pki`.
> Specify `password` only if `method` is set to `password`.

All properties that contain values are required, as described here:

`id`

A user-defined alphanumeric ID that uniquely identifies a collection of devices called a *device pool*. Devices that belong to a pool must have identical hardware. When you run a suite of tests, devices in the pool are used to parallelize the workload. Multiple devices are used to run different tests.

`sku`

An alphanumeric value that uniquely identifies the device under test. The SKU is used to track qualified boards.

> **Note**
>
> If you want to list your device in the AWS Partner Device Catalog, the SKU you specify here must match the SKU that you use in the listing process.

`features`

An array that contains the device's supported features. All features are required.

`arch`

The supported operating system architectures that the test run validates. Valid values are:

- `x86_64`
- `armv6l`
- `armv7l`
- `aarch64`

`docker`

Validates that the device meets all required technical dependencies to use the AWS-provided Docker application manager (`aws.greengrass.DockerApplicationManager`) component.

Enabling this feature also validates that the device can download a Docker container image from Amazon ECR . .

Valid values are any combination of `yes` or no.

`hsi`

Validates that the device can authenticate connections to the AWS IoT and AWS IoT Greengrass services using a private key and certificate that are stored in a hardware security module (HSM). This test also verifies that the AWS-provided PKCS#11 provider

component can interface with the HSM using a vendor-provided PKCS#11 library. For more information, see Hardware security integration.

Valid values are `hsm` or `no`.

> **Note**
>
> Testing the `hsi` is available only with IDT v4.9.3 and later versions.

`devices.id`

A user-defined unique identifier for the device being tested.

`devices.operatingSystem`

The device operating system. Supported values are `Linux` and `Windows`.

`connectivity.protocol`

The communication protocol used to communicate with this device. Currently, the only supported value is `ssh` for physical devices.

`connectivity.ip`

The IP address of the device being tested.

This property applies only if `connectivity.protocol` is set to `ssh`.

`connectivity.port`

Optional. The port number to use for SSH connections.

The default value is 22.

This property applies only if `connectivity.protocol` is set to `ssh`.

`connectivity.publicKeyPath`

Optional. The full path to the public key used to authenticate connections to the device under test.

When you specify the `publicKeyPath`, IDT validates the device's public key when it establishes an SSH connection to the device under test. If this value is not specified, IDT creates an SSH connection, but doesn't validate the device's public key.

We strongly recommend that you specify the path to the public key, and that you use a secure method to fetch this public key. For standard command line-based SSH clients, the public key is provided in the `known_hosts` file. If you specify a separate public key file, this file must use the same format as the `known_hosts` file, that is, *`ip-address key-type public-key`*. If there are multiple entries with the same ip-address, the entry for the key-type used by IDT must be before the other entries in the file.

`connectivity.auth`

Authentication information for the connection.

This property applies only if `connectivity.protocol` is set to `ssh`.

`connectivity.auth.method`

The authentication method used to access a device over the given connectivity protocol.

Supported values are:

- `pki`
- `password`

`connectivity.auth.credentials`

The credentials used for authentication.

`connectivity.auth.credentials.password`

The password used for signing in to the device being tested.

This value applies only if `connectivity.auth.method` is set to `password`.

`connectivity.auth.credentials.privKeyPath`

The full path to the private key used to sign in to the device under test.

This value applies only if `connectivity.auth.method` is set to `pki`.

`connectivity.auth.credentials.user`

The user name for signing in to the device being tested.

Configure userdata.json

IDT for AWS IoT Greengrass V2 also needs additional information about the location of test artifacts and AWS IoT Greengrass software.

You must provide this information using the `userdata.json` template located in *<device_tester_extract_location>*`/configs/userdata.json`:

```
{
    "TempResourcesDirOnDevice": "/path/to/temp/folder",
    "InstallationDirRootOnDevice": "/path/to/installation/folder",
    "GreengrassNucleusZip": "/path/to/aws.greengrass.nucleus.zip",
    "PreInstalled": "yes/no",
    "GreengrassV2TokenExchangeRole": "custom-iam-role-name",
 "hsm": {
        "greengrassPkcsPluginJar": "/path/to/aws.greengrass.crypto.Pkcs11Provider-
latest.jar",
        "pkcs11ProviderLibrary": "/path/to/pkcs11-vendor-library",
        "slotId": "slot-id",
        "slotLabel": "slot-label",
        "slotUserPin": "slot-pin",
        "keyLabel": "key-label",
        "preloadedCertificateArn": "certificate-arn"
        "rootCA": "path/to/root-ca"
    }
}
```

All properties that contain values are required as described here:

`TempResourcesDirOnDevice`

The full path to a temporary folder on the device under test in which to store test artifacts. Make sure that sudo permissions are not required to write to this directory.

> **Note**
>
> IDT deletes the contents of this folder when it finishes running a test.

`InstallationDirRootOnDevice`

The full path to a folder on the device in which to install AWS IoT Greengrass. For PreInstalled Greengrass, this is the path to the Greengrass installation directory.

You must set the required file permissions for this folder. Run the following command for each folder in the installation path.

```
sudo chmod 755 folder-name
```

`GreengrassNucleusZip`

The full path to the Greengrass nucleus ZIP (`greengrass-nucleus-latest.zip`) file on your host computer. This field is not required for testing with PreInstalled Greengrass.

> **Note**
>
> For information about the supported versions of the Greengrass nucleus for IDT for AWS IoT Greengrass, see Latest IDT version for AWS IoT Greengrass V2. To download the latest Greengrass software, see Download the AWS IoT Greengrass software.

`PreInstalled`

This feature is available for IDT v4.5.8 and later versions on Linux devices only.

(Optional) When the value is *yes*, IDT will assume the path mentioned in `InstallationDirRootOnDevice` to be the directory where Greengrass is installed.

For more information about how to install Greengrass on your device, see Install AWS IoT Greengrass Core software with automatic resource provisioning. If installing with manual provisioning, include the "Add the AWS IoT thing to a new or existing thing group" step when creating an AWS IoT thing manually. IDT assumes that the thing and thing group are created during installation setup. Make sure that these values are reflected in the `effectiveConfig.yaml` file. IDT checks for the file `effectiveConfig.yaml` under `<InstallationDirRootOnDevice>/config/effectiveConfig.yaml`.

For running tests with HSM, make sure that the `aws.greengrass.crypto.Pkcs11Provider` field is updated in `effectiveConfig.yaml`.

`GreengrassV2TokenExchangeRole`

(Optional) The custom IAM role that you want to use as the token exchange role that the device under test assumes to interact with AWS resources.

> **Note**
>
> IDT uses this custom IAM role instead of creating the default token exchange role during the test run. If you use a custom role, you can update the IAM permissions for the test

user to exclude the `iamResourcesUpdate` statement that allows the user to create and delete IAM roles and policies.

For more information about creating a custom IAM role as your token exchange role, see Configure a custom token exchange role.

`hsm`

This feature is available for IDT v4.5.1 and later.

(Optional) The configuration information for testing with an AWS IoT Greengrass Hardware Security Module (HSM). Otherwise, the `hsm` property should be omitted. For more information, see Hardware security integration.

This property applies only if `connectivity.protocol` is set to `ssh`.

Warning

The HSM configuration may be considered sensitive data if the hardware security module is shared between IDT and another system. In this situation, you may avoid securing these configuration values in plaintext by storing them in an AWS Parameter Store SecureString parameter and configuring IDT to fetch them during test execution. For more information, see ???

`hsm.greengrassPkcsPluginJar`

The full path to the PKCS#11 provider component that you download to the IDT host machine. AWS IoT Greengrass provides this component as JAR file that you can download to specify as a provisioning plugin during installation. You can download the latest version of the component's JAR file as the following URL: https://d2s8p88vqu9w66.cloudfront.net/releases/Pkcs11Provider/aws.greengrass.crypto.Pkcs11Provider-latest.jar.

`hsm.pkcs11ProviderLibrary`

The full path to the PKCS#11 library that is provided by the hardware security module (HSM) vendor to interact with the HSM.

`hsm.slotId`

The slot ID that is used to identify the HSM slot to which you load the key and certificate.

`hsm.slotLabel`

The slot label that is used to identify the HSM slot to which you load the key and certificate.

`hsm.slotUserPin`

The user PIN that IDT uses to authenticate AWS IoT Greengrass Core software to the HSM.

> **Note**
>
> As a security best practice, don't use the same user PIN on production devices.

`hsm.keyLabel`

The label used to identify the key in the hardware module. Both the key and the certificate must use the same key label.

`hsm.preloadedCertificateArn`

The Amazon Resource Name (ARN) of the uploaded device certificate in the AWS IoT cloud.

You must have previously generated this certificate using the key in the HSM, imported it into your HSM, and uploaded it to the AWS IoT cloud. For information about generating and importing the certificate, see the documentation for your HSM.

You must upload the certificate to the same account and Region that you provide in config.json.. For more information about uploading your certificate to AWS IoT, see Register a client certificate manually in the *AWS IoT Developer Guide*.

`hsm.rootCAPath`

(Optional) The full path on the IDT host machine to the root certificate authority (CA) that signed your certificate. This is required if the certificate in your HSM created is not signed by the Amazon root CA.

Fetch configuration from AWS Parameter Store

AWS IoT Device Tester (IDT) includes an optional feature to fetch configuration values from the AWS Systems Manager Parameter Store. AWS Parameter Store allows for secure and encrypted storage of configurations. When configured, IDT can fetch parameters from AWS Parameter Store

in place of storing parameters in plaintext inside the `userdata.json` file. This is useful for any sensitive data that should be stored encrypted, such as: passwords, pins, and other secrets.

1. To use this feature, you must update the permissions used in creating your IDT user to allow the GetParameter action on the parameters that IDT is configured to use. The below is an example of a permission statement that can be added to the IDT user. For more information, see AWS Systems Manager userguide.

   ```
   {
          "Sid":"parameterStoreResources",
          "Effect": "Allow",
           "Action": [
               "ssm:GetParameter"
           ],
           "Resource": "arn:aws:ssm:*:*:parameter/IDT*"
   }
   ```

   The above permission is configured to allow fetching all parameters with a name beginning with `IDT`, by using the wildcard character `*`. You should customize this to your needs so IDT has access to fetch any configured parameters based on the naming of the parameters you are using.

2. You need to store your configuration values inside AWS Paramater Store. This can be done from the AWS console or from the AWS CLI. AWS Parameter Store allows you to choose encrypted or unencrypted storage. For storage of sensitive values like secrets, passwords, and pins, you should use the encrypted option which is a parameter type of SecureString. To upload a parameter using the AWS CLI, you can use the following command:

   ```
   aws ssm put-parameter --name IDT-example-name --value IDT-example-value --type
    SecureString
   ```

   You can verify that a parameter is stored using the following command. (Optional) Use the `--with-decryption` flag to fetch a decrypted SecureString parameter.

   ```
   aws ssm get-parameter --name IDT-example-name
   ```

   Using the AWS CLI will upload the parameter in the AWS region of the current CLI user and IDT will fetch parameters from the region configured in `config.json`. To check your region from the AWS CLI, use the following:

```
aws configure get region
```

3. Once you have a configuration value in the AWS Cloud, you can update any value inside the IDT configuration to fetch from the AWS Cloud. To do so, you use a placeholder in your IDT configuration of the form `{{AWS.Parameter.parameter_name}}` to fetch the parameter by that name from the AWS Parameter Store.

   For example, suppose you want to use the `IDT-example-name` parameter from Step 2 as the HSM keyLabel in your HSM configuration. To do this, you can update your `userdata.json` as follows:

   ```
   "hsm": {
           "keyLabel": "{{AWS.Parameter.IDT-example-name}}",
           [...]
       }
   ```

   IDT will fetch the value of this parameter at runtime that was set to `IDT-example-value` in Step 2. This configuration is similar to setting `"keyLabel": "IDT-example-value"` but, instead, that value is store as encrypted in the AWS Cloud.

Run the AWS IoT Greengrass qualification suite

After you set the required configuration, you can start the tests. The runtime of the full test suite depends on your hardware. For reference, it takes approximately 30 minutes to complete the full test suite on a Raspberry Pi 3B.

Use the following `run-suite` command to run a suite of tests.

```
devicetester_[linux | mac | win]_x86-64 run-suite  \\
    --suite-id suite-id  \\
    --group-id group-id  \\
    --pool-id your-device-pool \\
    --test-id test-id  \\
    --update-idt y|n  \\
    --userdata userdata.json
```

All options are optional. For example, you can omit `pool-id` if you have only one device pool, which is a set of identical devices, defined in your `device.json` file. Or, you can omit `suite-id` if you want to run the latest test suite version in the `tests` folder.

> **Note**
>
> IDT prompts you if a newer test suite version is available online. For more information, see the section called "Test suite versions".

Example commands to run the qualification suite

The following command line examples show you how to run the qualification tests for a device pool. For more information about `run-suite` and other IDT commands, see the section called "IDT commands".

Use the following command to run all test groups in a specified test suite. The `list-suites` command lists the test suites that are in the `tests` folder.

```
devicetester_[linux | mac | win]_x86-64 run-suite \
    --suite-id GGV2Q_1.0.0 \
    --pool-id <pool-id> \
    --userdata userdata.json
```

Use the following command to run a specific test group in a test suite. The `list-groups` command lists the test groups in a test suite.

```
devicetester_[linux | mac | win]_x86-64 run-suite \
    --suite-id GGV2Q_1.0.0 \
    --group-id <group-id> \
    --pool-id <pool-id> \
    --userdata userdata.json
```

Use the following command to run a specific test case in a test group.

```
devicetester_[linux | mac | win]_x86-64 run-suite \
    --group-id <group-id> \
    --test-id <test-id> \
    --userdata userdata.json
```

Use the following command to run multiple test cases in a test group.

```
devicetester_[linux | mac | win]_x86-64 run-suite \
    --group-id <group-id> \
    --test-id <test-id1>,<test-id2>
    --userdata userdata.json
```

Use the following command to list all of the test cases in a test group.

```
devicetester_[linux | mac | win]_x86-64 list-test-cases --group-id <group-id>
```

We recommend that you run the full qualification test suite, which runs test group dependencies in the correct order. If you choose to run specific test groups, we recommend that you first run the dependency checker test group to make sure all Greengrass dependencies are installed before you run related test groups. For example:

- Run `coredependencies` before running core qualification test groups.

IDT for AWS IoT Greengrass V2 commands

The IDT commands are located in the *<device-tester-extract-location>*`/bin` directory. To run a test suite, you provide the command in the following format:

`help`

Lists information about the specified command.

`list-groups`

Lists the groups in a given test suite.

`list-suites`

Lists the available test suites.

`list-supported-products`

Lists the supported products, in this case AWS IoT Greengrass versions, and test suite versions for the current IDT version.

`list-test-cases`

Lists the test cases in a given test group. The following option is supported:

- `group-id`. The test group to search for. This option is required and must specify a single group.

`run-suite`

Runs a suite of tests on a pool of devices. The following are some supported options:

- `suite-id`. The test suite version to run. If not specified, IDT uses the latest version in the `tests` folder.
- `group-id`. The test groups to run, as a comma-separated list. If not specified, IDT runs all appropriate test groups in the test suite depending on the configured settings in `device.json`. IDT doesn't run any test groups that the device doesn't support based on your configured settings, even if those test groups are specified in the `group-id` list.
- `test-id`. The test cases to run, as a comma-separated list. When specified, `group-id` must specify a single group.
- `pool-id`. The device pool to test. You must specify a pool if you have multiple device pools defined in your `device.json` file.
- `stop-on-first-failure`. Configures IDT to stop running on the first failure. Use this option with `group-id` when you want to debug the specified test groups. Do not use this option when running a full test-suite to generate a qualification report.
- `update-idt`. Sets the response for the prompt to update IDT. The `Y` response stops the test execution if IDT detects there is a newer version. The `N` response continues the test execution.
- `userdata`. The full path to the `userdata.json` file that contains information about test artifact paths. This option is required for the `run-suite` command. The `userdata.json` file must be located in the *`devicetester_extract_location`*/devicetester_ggv2_*`[win|mac|linux]`*/configs/ directory.

For more information about `run-suite` options, use the `help` option:

```
devicetester_[linux | mac | win]_x86-64 run-suite -h
```

Understanding results and logs

This section describes how to view and interpret IDT result reports and logs.

To troubleshoot errors, see Troubleshooting IDT for AWS IoT Greengrass V2.

Viewing results

While running, IDT writes errors to the console, log files, and test reports. After IDT completes the qualification test suite, it generates two test reports. These reports are located in *<device-tester-extract-location>*/results/*<execution-id>*/. Both reports capture the results from running the qualification test suite.

The awsiotdevicetester_report.xml is the qualification test report that you submit to AWS to list your device in the AWS Partner Device Catalog. The report contains the following elements:

- The IDT version.
- The AWS IoT Greengrass version that was tested.
- The SKU and the device pool name specified in the device.json file.
- The features of the device pool specified in the device.json file.
- The aggregate summary of test results.
- A breakdown of test results by libraries that were tested based on the device features, such as local resource access, shadow, and MQTT.

The GGV2Q_Result.xml report is in JUnit XML format. You can integrate it into continuous integration and deployment platforms like Jenkins, Bamboo, and so on. The report contains the following elements:

- Aggregate summary of test results.
- Breakdown of test results by the AWS IoT Greengrass functionality that was tested.

Interpreting AWS IoT Device Tester results

The report section in awsiotdevicetester_report.xml or awsiotdevicetester_report.xml lists the tests that were run and the results.

The first XML tag <testsuites> contains the summary of the test run. For example:

```
<testsuites name="GGQ results" time="2299" tests="28" failures="0" errors="0"
 disabled="0">
```

Attributes used in the <testsuites> tag

`name`

The name of the test suite.

`time`

The time, in seconds, that it took to run the qualification suite.

`tests`

The number of tests that were run.

`failures`

The number of tests that were run, but did not pass.

`errors`

The number of tests that IDT couldn't run.

`disabled`

Ignore this attribute. It is not used.

The `awsiotdevicetester_report.xml` file contains an `<awsproduct>` tag that contains information about the product being tested and the product features that were validated after running a suite of tests.

Attributes used in the <awsproduct> tag

`name`

The name of the product being tested.

`version`

The version of the product being tested.

`features`

The features validated. Features marked as `required` are required to submit your board for qualification. The following snippet shows how this information appears in the `awsiotdevicetester_report.xml` file.

```
<name="aws-iot-greengrass-v2-core" value="supported" type="required"></feature>
```

If there are no test failures or errors for the required features, your device meets the technical requirements to run AWS IoT Greengrass and can interoperate with AWS IoT services. If you want to list your device in the AWS Partner Device Catalog, you can use this report as qualification evidence.

In the event of test failures or errors, you can identify the test that failed by reviewing the `<testsuites>` XML tags. The `<testsuite>` XML tags inside the `<testsuites>` tag show the test result summary for a test group. For example:

```
<testsuite name="combination" package="" tests="1" failures="0" time="161" disabled="0"
 errors="0" skipped="0">
```

The format is similar to the `<testsuites>` tag, but with a `skipped` attribute that is not used and can be ignored. Inside each `<testsuite>` XML tag, there are `<testcase>` tags for each test that was run for a test group. For example:

```
<testcase classname="Security Combination (IPD + DCM) Test Context" name="Security
 Combination IP Change Tests sec4_test_1: Should rotate server cert when IPD disabled
 and following changes are made:Add CIS conn info and Add another CIS conn info"
 attempts="1"></testcase>>
```

Attributes used in the `<testcase>` tag

`name`

The name of the test.

`attempts`

The number of times IDT ran the test case.

When a test fails or an error occurs, `<failure>` or `<error>` tags are added to the `<testcase>` tag with information for troubleshooting. For example:

```
<testcase classname="mcu.Full_MQTT" name="AFQP_MQTT_Connect_HappyCase" attempts="1">
 <failure type="Failure">Reason for the test failure</failure>
 <error>Reason for the test execution error</error>
```

```
</testcase>
```

Viewing logs

IDT generates logs from test runs in *<devicetester-extract-location>*/results/*<execution-id>*/logs. Two sets of logs are generated:

`test_manager.log`

Logs generated from the Test Manager component of AWS IoT Device Tester (for example, logs related to configuration, test sequencing, and report generation).

<test-case-id>`.log (for example, lambdaDeploymentTest.log)`

Logs of the test case within the test group, including logs from the device under test. Starting with IDT v4.2.0, IDT groups the test logs for each test case in a separate *<test-case-id>* folder within the *<devicetester-extract-location>*/results/*<execution-id>*/logs/*<test-group-id>/* directory.

Use IDT to develop and run your own test suites

Starting in IDT v4.0.1, IDT for AWS IoT Greengrass V2 combines a standardized configuration setup and result format with a test suite environment that enables you to develop custom test suites for your devices and device software. You can add custom tests for your own internal validation or provide them to your customers for device verification.

Use IDT to develop and run custom test suites, as follows:

To develop custom test suites

- Create test suites with custom test logic for the Greengrass device that you want to test.
- Provide IDT with your custom test suites to test runners. Include information about specific settings configurations for your test suites.

To run custom test suites

- Set up the device that you want to test.
- Implement the settings configurations as required by the test suites that you want to use.
- Use IDT to run your custom test suites.
- View the test results and execution logs for the tests run by IDT.

Download the latest version of AWS IoT Device Tester for AWS IoT Greengrass

Download the latest version of IDT and extract the software into a location (*<device-tester-extract-location>*) on your file system where you have read/write permissions.

> **Note**
>
> IDT does not support being run by multiple users from a shared location, such as an NFS directory or a Windows network shared folder. We recommend that you extract the IDT package to a local drive and run the IDT binary on your local workstation.
> Windows has a path length limitation of 260 characters. If you are using Windows, extract IDT to a root directory like `C:\` or `D:\` to keep your paths under the 260 character limit.

Test suite creation workflow

Test suites are composed of three types of files:

- Configuration files that provide IDT with information about how to run the test suite.
- Test executable files that IDT uses to run test cases.
- Additional files required to run tests.

Complete the following basic steps to create custom IDT tests:

1. Create configuration files for your test suite.
2. Create test case executables that contain the test logic for your test suite.
3. Verify and document the configuration information required for test runners to run the test suite.
4. Verify that IDT can run your test suite and produce test results as expected.

To quickly build a sample custom suite and run it, follow the instructions in Tutorial: Build and run the sample IDT test suite.

To get started creating a custom test suite in Python, see Tutorial: Develop a simple IDT test suite.

Tutorial: Build and run the sample IDT test suite

The AWS IoT Device Tester download includes the source code for a sample test suite. You can complete this tutorial to build and run the sample test suite to understand how you can use IDT for AWS IoT Greengrass to run custom test suites.

In this tutorial, you will complete the following steps:

1. Build the sample test suite
2. Use IDT to run the sample test suite

Prerequisites

To complete this tutorial, you need the following:

- **Host computer requirements**
  - Latest version of AWS IoT Device Tester
  - Python 3.7 or later

    To check the version of Python installed on your computer, run the following command:

    ```
    python3 --version
    ```

    On Windows, if using this command returns an error, then use `python --version` instead. If the returned version number is 3.7 or greater, then run the following command in a Powershell terminal to set `python3` as an alias for your `python` command.

    ```
    Set-Alias -Name "python3" -Value "python"
    ```

    If no version information is returned or if the version number is less than 3.7, follow the instructions in Downloading Python to install Python 3.7+. For more information, see the Python documentation.
  - urllib3

    To verify that `urllib3` is installed correctly, run the following command:

    ```
    python3 -c 'import urllib3'
    ```

If `urllib3` is not installed, run the following command to install it:

```
python3 -m pip install urllib3
```

- **Device requirements**
  - A device with a Linux operating system and a network connection to the same network as your host computer.

    We recommend that you use a Raspberry Pi with Raspberry Pi OS. Make sure you set up SSH on your Raspberry Pi to remotely connect to it.

Configure device information for IDT

Configure your device information for IDT to run the test. You must update the `device.json` template located in the *<device-tester-extract-location>*`/configs` folder with the following information.

```
[
  {
    "id": "pool",
    "sku": "N/A",
    "devices": [
      {
        "id": "<device-id>",
        "connectivity": {
          "protocol": "ssh",
          "ip": "<ip-address>",
          "port": "<port>",
          "auth": {
            "method": "pki | password",
            "credentials": {
              "user": "<user-name>",
              "privKeyPath": "/path/to/private/key",
              "password": "<password>"
            }
          }
        }
      }
    ]
  }
```

```
]
```

In the `devices` object, provide the following information:

`id`

A user-defined unique identifier for your device.

`connectivity.ip`

The IP address of your device.

`connectivity.port`

Optional. The port number to use for SSH connections to your device.

`connectivity.auth`

Authentication information for the connection.

This property applies only if `connectivity.protocol` is set to `ssh`.

`connectivity.auth.method`

The authentication method used to access a device over the given connectivity protocol.

Supported values are:

- `pki`
- `password`

`connectivity.auth.credentials`

The credentials used for authentication.

`connectivity.auth.credentials.user`

The user name used to sign in to your device.

`connectivity.auth.credentials.privKeyPath`

The full path to the private key used to sign in to your device.

This value applies only if `connectivity.auth.method` is set to `pki`.

`devices.connectivity.auth.credentials.password`

The password used for signing in to your device.

This value applies only if `connectivity.auth.method` is set to `password`.

> **Note**
>
> Specify `privKeyPath` only if `method` is set to `pki`.
> Specify `password` only if `method` is set to `password`.

Build the sample test suite

The *<device-tester-extract-location>*`/samples/python` folder contains sample configuration files, source code, and the IDT Client SDK that you can combine into a test suite using the provided build scripts. The following directory tree shows the location of these sample files:

```
<device-tester-extract-location>
### ...
### tests
### samples
#   ### ...
#   ### python
#       ### configuration
#       ### src
#       ### build-scripts
#           ### build.sh
#           ### build.ps1
### sdks
    ### ...
    ### python
        ### idt_client
```

To build the test suite, run the following commands on your host computer:

Windows

```
cd <device-tester-extract-location>/samples/python/build-scripts
./build.ps1
```

Linux, macOS, or UNIX

```
cd <device-tester-extract-location>/samples/python/build-scripts
```

```
./build.sh
```

This creates the sample test suite in the `IDTSampleSuitePython_1.0.0` folder within the *<device-tester-extract-location>*`/tests` folder. Review the files in the `IDTSampleSuitePython_1.0.0` folder to understand how the sample test suite is structured, and to see various examples of test case executables and test configuration JSON files.

> **Note**
>
> The sample test suite includes python source code. Do not include sensitive information in your test suite code.

Next step: Use IDT to run the sample test suite that you created.

Use IDT to run the sample test suite

To run the sample test suite, run the following commands on your host computer:

```
cd <device-tester-extract-location>/bin
./devicetester_[linux | mac | win_x86-64] run-suite --suite-id IDTSampleSuitePython
```

IDT runs the sample test suite and streams the results to the console. When the test has finished running, you see the following information:

```
========== Test Summary ==========
Execution Time:         5s
Tests Completed:        4
Tests Passed:           4
Tests Failed:           0
Tests Skipped:          0
----------------------------------
Test Groups:
    sample_group:       PASSED
----------------------------------
Path to IoT Device Tester Report: /path/to/devicetester/
results/87e673c6-1226-11eb-9269-8c8590419f30/awsiotdevicetester_report.xml
Path to Test Execution Logs: /path/to/devicetester/
results/87e673c6-1226-11eb-9269-8c8590419f30/logs
```

```
Path to Aggregated JUnit Report: /path/to/devicetester/
results/87e673c6-1226-11eb-9269-8c8590419f30/IDTSampleSuitePython_Report.xml
```

Troubleshooting

Use the following information to help resolve any issues with completing the tutorial.

Test case does not run successfully

If the test does not run successfully, IDT streams the error logs to the console that can help you troubleshoot the test run. Make sure that you meet all the prerequisites for this tutorial.

Cannot connect to the device under test

Verify the following:

- Your `device.json` file contains the correct IP address, port, and authentication information.
- You can connect to your device over SSH from your host computer.

Tutorial: Develop a simple IDT test suite

A test suite combines the following:

- Test executables that contain the test logic
- Configuration files that describe the test suite

This tutorial shows you how to use IDT for AWS IoT Greengrass to develop a Python test suite that contains a single test case. In this tutorial, you will complete the following steps:

1. Create a test suite directory
2. Create configuration files
3. Create the test case executable
4. Run the test suite

Prerequisites

To complete this tutorial, you need the following:

- **Host computer requirements**
  - Latest version of AWS IoT Device Tester
  - Python 3.7 or later

    To check the version of Python installed on your computer, run the following command:

    ```
    python3 --version
    ```

    On Windows, if using this command returns an error, then use `python --version` instead. If the returned version number is 3.7 or greater, then run the following command in a Powershell terminal to set `python3` as an alias for your `python` command.

    ```
    Set-Alias -Name "python3" -Value "python"
    ```

    If no version information is returned or if the version number is less than 3.7, follow the instructions in Downloading Python to install Python 3.7+. For more information, see the Python documentation.
  - urllib3

    To verify that `urllib3` is installed correctly, run the following command:

    ```
    python3 -c 'import urllib3'
    ```

    If `urllib3` is not installed, run the following command to install it:

    ```
    python3 -m pip install urllib3
    ```
- **Device requirements**
  - A device with a Linux operating system and a network connection to the same network as your host computer.

    We recommend that you use a Raspberry Pi with Raspberry Pi OS. Make sure you set up SSH on your Raspberry Pi to remotely connect to it.

Create a test suite directory

IDT logically separates test cases into test groups within each test suite. Each test case must be inside a test group. For this tutorial, create a folder called `MyTestSuite_1.0.0` and create the following directory tree within this folder:

```
MyTestSuite_1.0.0
### suite
    ### myTestGroup
        ### myTestCase
```

Create configuration files

Your test suite must contain the following required configuration files:

Required configuration files

`suite.json`

Contains information about the test suite. See Configure suite.json.

`group.json`

Contains information about a test group. You must create a `group.json` file for each test group in your test suite. See Configure group.json.

`test.json`

Contains information about a test case. You must create a `test.json` file for each test case in your test suite. See Configure test.json.

1. In the `MyTestSuite_1.0.0/suite` folder, create a `suite.json` file with the following structure:

   ```
   {
       "id": "MyTestSuite_1.0.0",
       "title": "My Test Suite",
       "details": "This is my test suite.",
       "userDataRequired": false
   }
   ```

2. In the `MyTestSuite_1.0.0/myTestGroup` folder, create a `group.json` file with the following structure:

```
{
    "id": "MyTestGroup",
    "title": "My Test Group",
    "details": "This is my test group.",
    "optional": false
}
```

3. In the `MyTestSuite_1.0.0/myTestGroup/myTestCase` folder, create a `test.json` file with the following structure:

```
{
    "id": "MyTestCase",
    "title": "My Test Case",
    "details": "This is my test case.",
    "execution": {
        "timeout": 300000,
        "linux": {
            "cmd": "python3",
            "args": [
                "myTestCase.py"
            ]
        },
        "mac": {
            "cmd": "python3",
            "args": [
                "myTestCase.py"
            ]
        },
        "win": {
            "cmd": "python3",
            "args": [
                "myTestCase.py"
            ]
        }
    }
}
```

The directory tree for your `MyTestSuite_1.0.0` folder should now look like the following:

```
MyTestSuite_1.0.0
### suite
```

```
    ### suite.json
    ### myTestGroup
        ### group.json
        ### myTestCase
            ### test.json
```

Get the IDT client SDK

You use the IDT client SDK to enable IDT to interact with the device under test and to report test results. For this tutorial, you will use the Python version of the SDK.

From the *<device-tester-extract-location>*/sdks/python/ folder, copy the `idt_client` folder to your `MyTestSuite_1.0.0/suite/myTestGroup/myTestCase` folder.

To verify that the SDK was successfully copied, run the following command.

```
cd MyTestSuite_1.0.0/suite/myTestGroup/myTestCase
python3 -c 'import idt_client'
```

Create the test case executable

Test case executables contain the test logic that you want to run. A test suite can contain multiple test case executables. For this tutorial, you will create only one test case executable.

1. Create the test suite file.

   In the `MyTestSuite_1.0.0/suite/myTestGroup/myTestCase` folder, create a `myTestCase.py` file with the following content:

   ```
   from idt_client import *

   def main():
       # Use the client SDK to communicate with IDT
       client = Client()

   if __name__ == "__main__":
       main()
   ```

2. Use client SDK functions to add the following test logic to your `myTestCase.py` file:

   a. Run an SSH command on the device under test.

```
from idt_client import *

def main():
    # Use the client SDK to communicate with IDT
    client = Client()

    # Create an execute on device request
    exec_req = ExecuteOnDeviceRequest(ExecuteOnDeviceCommand("echo 'hello
 world'"))

    # Run the command
    exec_resp = client.execute_on_device(exec_req)

    # Print the standard output
    print(exec_resp.stdout)

if __name__ == "__main__":
    main()
```

b. Send the test result to IDT.

```
from idt_client import *

def main():
    # Use the client SDK to communicate with IDT
    client = Client()

    # Create an execute on device request
    exec_req = ExecuteOnDeviceRequest(ExecuteOnDeviceCommand("echo 'hello
 world'"))

    # Run the command
    exec_resp = client.execute_on_device(exec_req)

    # Print the standard output
    print(exec_resp.stdout)

    # Create a send result request
    sr_req = SendResultRequest(TestResult(passed=True))

    # Send the result
    client.send_result(sr_req)
```

```
if __name__ == "__main__":
    main()
```

Configure device information for IDT

Configure your device information for IDT to run the test. You must update the `device.json` template located in the *<device-tester-extract-location>*`/configs` folder with the following information.

```
[
  {
    "id": "pool",
    "sku": "N/A",
    "devices": [
      {
        "id": "<device-id>",
        "connectivity": {
          "protocol": "ssh",
          "ip": "<ip-address>",
          "port": "<port>",
          "auth": {
            "method": "pki | password",
            "credentials": {
              "user": "<user-name>",
              "privKeyPath": "/path/to/private/key",
              "password": "<password>"
            }
          }
        }
      }
    ]
  }
]
```

In the `devices` object, provide the following information:

`id`

A user-defined unique identifier for your device.

`connectivity.ip`

The IP address of your device.

`connectivity.port`

Optional. The port number to use for SSH connections to your device.

`connectivity.auth`

Authentication information for the connection.

This property applies only if `connectivity.protocol` is set to `ssh`.

`connectivity.auth.method`

The authentication method used to access a device over the given connectivity protocol.

Supported values are:

- `pki`
- `password`

`connectivity.auth.credentials`

The credentials used for authentication.

`connectivity.auth.credentials.user`

The user name used to sign in to your device.

`connectivity.auth.credentials.privKeyPath`

The full path to the private key used to sign in to your device.

This value applies only if `connectivity.auth.method` is set to `pki`.

`devices.connectivity.auth.credentials.password`

The password used for signing in to your device.

This value applies only if `connectivity.auth.method` is set to `password`.

Note

Specify `privKeyPath` only if `method` is set to `pki`.

Specify `password` only if `method` is set to `password`.

Run the test suite

After you create your test suite, you want to make sure that it functions as expected. Complete the following steps to run the test suite with your existing device pool to do so.

1. Copy your `MyTestSuite_1.0.0` folder into *<device-tester-extract-location>*/`tests`.
2. Run the following commands:

```
cd <device-tester-extract-location>/bin
./devicetester_[linux | mac | win_x86-64] run-suite --suite-id MyTestSuite
```

IDT runs your test suite and streams the results to the console. When the test has finished running, you see the following information:

```
time="2020-10-19T09:24:47-07:00" level=info msg=Using pool: pool
time="2020-10-19T09:24:47-07:00" level=info msg=Using test suite "MyTestSuite_1.0.0"
 for execution
time="2020-10-19T09:24:47-07:00" level=info msg=b'hello world\n'
 suiteId=MyTestSuite groupId=myTestGroup testCaseId=myTestCase deviceId=my-device
 executionId=9a52f362-1227-11eb-86c9-8c8590419f30
time="2020-10-19T09:24:47-07:00" level=info msg=All tests finished.
 executionId=9a52f362-1227-11eb-86c9-8c8590419f30
time="2020-10-19T09:24:48-07:00" level=info msg=

========== Test Summary ==========
Execution Time:         1s
Tests Completed:        1
Tests Passed:           1
Tests Failed:           0
Tests Skipped:          0
----------------------------------
Test Groups:
    myTestGroup:        PASSED
----------------------------------
Path to IoT Device Tester Report: /path/to/devicetester/
results/9a52f362-1227-11eb-86c9-8c8590419f30/awsiotdevicetester_report.xml
```

```
Path to Test Execution Logs: /path/to/devicetester/
results/9a52f362-1227-11eb-86c9-8c8590419f30/logs
Path to Aggregated JUnit Report: /path/to/devicetester/
results/9a52f362-1227-11eb-86c9-8c8590419f30/MyTestSuite_Report.xml
```

Troubleshooting

Use the following information to help resolve any issues with completing the tutorial.

Test case does not run successfully

If the test does not run successfully, IDT streams the error logs to the console that can help you troubleshoot the test run. Before you check the error logs, verify the following:

- The IDT client SDK is in the correct folder as described in this step.
- You meet all the prerequisites for this tutorial.

Cannot connect to the device under test

Verify the following:

- Your `device.json` file contains the correct IP address, port, and authentication information.
- You can connect to your device over SSH from your host computer.

Create IDT test suite configuration files

This section describes the formats in which you create configuration files that you include when you write a custom test suite.

Required configuration files

`suite.json`

Contains information about the test suite. See Configure suite.json.

`group.json`

Contains information about a test group. You must create a `group.json` file for each test group in your test suite. See Configure group.json.

`test.json`

Contains information about a test case. You must create a `test.json` file for each test case in your test suite. See Configure test.json.

Optional configuration files

`test_orchestrator.yaml` or `state_machine.json`

Defines how tests are run when IDT runs the test suite. SSe Configure test_orchestrator.yaml.

> **Note**
>
> Starting in IDT v4.5.1, you use the `test_orchestrator.yaml` file to define the test workflow. In previous versions of IDT, you use the `state_machine.json` file. For information about the state machine, see Configure the IDT state machine.

`userdata_schema.json`

Defines the schema for the `userdata.json` file that test runners can include in their setting configuration. The `userdata.json` file is used for any additional configuration information that is required to run the test but is not present in the `device.json` file. See Configure userdata_schema.json.

Configuration files are placed in your *<custom-test-suite-folder>* as shown here.

```
<custom-test-suite-folder>
### suite
    ### suite.json
    ### test_orchestrator.yaml
    ### userdata_schema.json
    ### <test-group-folder>
        ### group.json
        ### <test-case-folder>
            ### test.json
```

Configure suite.json

The `suite.json` file sets environment variables and determines whether user data is required to run the test suite. Use the following template to configure your *<custom-test-suite-folder>*`/suite/suite.json` file:

```
{
    "id": "<suite-name>_<suite-version>",
    "title": "<suite-title>",
    "details": "<suite-details>",
    "userDataRequired": true | false,
    "environmentVariables": [
        {
            "key": "<name>",
            "value": "<value>",
        },
        ...
        {
            "key": "<name>",
            "value": "<value>",
        }
    ]
}
```

All fields that contain values are required as described here:

`id`

A unique user-defined ID for the test suite. The value of `id` must match the name of the test suite folder in which the `suite.json` file is located. The suite name and suite version must also meet the following requirements:

- *<suite-name>* cannot contain underscores.
- *<suite-version>* is denoted as *x.x.x*, where x is a number.

The ID is shown in IDT-generated test reports.

`title`

A user-defined name for the product or feature being tested by this test suite. The name is displayed in the IDT CLI for test runners.

`details`

A short description of the purpose of the test suite.

`userDataRequired`

Defines whether test runners need to include custom information in a `userdata.json` file. If you set this value to `true`, you must also include the `userdata_schema.json` file in your test suite folder.

`environmentVariables`

Optional. An array of environment variables to set for this test suite.

`environmentVariables.key`

The name of the environment variable.

`environmentVariables.value`

The value of the environment variable.

Configure group.json

The `group.json` file defines whether a test group is required or optional. Use the following template to configure your *<custom-test-suite-folder>*/suite/*<test-group>*/`group.json` file:

```
{
    "id": "<group-id>",
    "title": "<group-title>",
    "details": "<group-details>",
    "optional": true | false,
}
```

All fields that contain values are required as described here:

`id`

A unique user-defined ID for the test group. The value of `id` must match the name of the test group folder in which the `group.json` file is located, and can't contain underscores (_). The ID is used in IDT-generated test reports.

`title`

A descriptive name for the test group. The name is displayed in the IDT CLI for test runners.

`details`

A short description of the purpose of the test group.

`optional`

Optional. Set to `true` to display this test group as an optional group after IDT finishes running required tests. Default value is `false`.

Configure test.json

The `test.json` file determines the test case executables and the environment variables that are used by a test case. For more information about creating test case executables, see Create IDT test case executables.

Use the following template to configure your *<custom-test-suite-folder>*/suite/*<test-group>*/*<test-case>*/test.json file:

```
{
    "id": "<test-id>",
    "title": "<test-title>",
    "details": "<test-details>",
    "requireDUT": true | false,
    "requiredResources": [
        {
            "name": "<resource-name>",
            "features": [
                {
                    "name": "<feature-name>",
                    "version": "<feature-version>",
                    "jobSlots": <job-slots>
                }
            ]
        }
    ],
    "execution": {
        "timeout": <timeout>,
        "mac": {
            "cmd": "/path/to/executable",
            "args": [
```

```
                "<argument>"
            ],
        },
        "linux": {
            "cmd": "/path/to/executable",
            "args": [
                "<argument>"
            ],
        },
        "win": {
            "cmd": "/path/to/executable",
            "args": [
                "<argument>"
            ]
        }
    },
    "environmentVariables": [
        {
            "key": "<name>",
            "value": "<value>",
        }
    ]
}
```

All fields that contain values are required as described here:

`id`

A unique user-defined ID for the test case. The value of `id` must match the name of the test case folder in which the `test.json` file is located, and can't contain underscores (_). The ID is used in IDT-generated test reports. .

`title`

A descriptive name for the test case. The name is displayed in the IDT CLI for test runners.

`details`

A short description of the purpose of the test case.

`requireDUT`

Optional. Set to `true` if a device is required to run this test, otherwise set to `false`. Default value is `true`. Test runners will configure the devices they will use to run the test in their `device.json` file.

`requiredResources`

Optional. An array that provides information about resource devices needed to run this test.

`requiredResources.name`

The unique name to give the resource device when this test is running.

`requiredResources.features`

An array of user-defined resource device features.

`requiredResources.features.name`

The name of the feature. The device feature for which you want to use this device. This name is matched against the feature name provided by the test runner in the `resource.json` file.

`requiredResources.features.version`

Optional. The version of the feature. This value is matched against the feature version provided by the test runner in the `resource.json` file. If a version is not provided, then the feature is not checked. If a version number is not required for the feature, leave this field blank.

`requiredResources.features.jobSlots`

Optional. The number of simultaneous tests that this feature can support. The default value is 1. If you want IDT to use distinct devices for individual features, then we recommend that you set this value to 1.

`execution.timeout`

The amount of time (in milliseconds) that IDT waits for the test to finish running. For more information about setting this value, see Create IDT test case executables.

`execution.os`

The test case executables to run based on the operating system of the host computer that runs IDT. Supported values are `linux`, `mac`, and `win`.

`execution.os.cmd`

The path to the test case executable that you want to run for the specified operating system. This location must be in the system path.

`execution.os.args`

Optional. The arguments to provide to run the test case executable.

`environmentVariables`

Optional. An array of environment variables set for this test case.

`environmentVariables.key`

The name of the environment variable.

`environmentVariables.value`

The value of the environment variable.

> **Note**
>
> If you specify the same environment variable in the `test.json` file and in the `suite.json` file, the value in the `test.json` file takes precedence.

Configure test_orchestrator.yaml

A test orchestrator is a construct that controls the test suite execution flow. It determines the starting state of a test suite, manages state transitions based on user-defined rules, and continues to transition through those states until it reaches the end state.

If your test suite doesn't include a user-defined test orchestrator, IDT will generate a test orchestrator for you.

The default test orchestrator performs the following functions:

- Provides test runners with the ability to select and run specific test groups, instead of the entire test suite.
- If specific test groups are not selected, runs every test group in the test suite in a random order.
- Generates reports and prints a console summary that shows the test results for each test group and test case.

For more information about how the IDT test orchestrator functions, see Configure the IDT test orchestrator.

Configure userdata_schema.json

The `userdata_schema.json` file determines the schema in which test runners provide user data. User data is required if your test suite requires information that is not present in the `device.json` file. For example, your tests might need Wi-Fi network credentials, specific open ports, or certificates that a user must provide. This information can be provided to IDT as an input parameter called `userdata`, the value for which is a `userdata.json` file, that users create in their *<device-tester-extract-location>*`/config` folder. The format of the `userdata.json` file is based on the `userdata_schema.json` file that you include in the test suite.

To indicate that test runners must provide a `userdata.json` file:

1. In the `suite.json` file, set `userDataRequired` to `true`.
2. In your *<custom-test-suite-folder>*, create a `userdata_schema.json` file.
3. Edit the `userdata_schema.json` file to create a valid IETF Draft v4 JSON Schema.

When IDT runs your test suite, it automatically reads the schema and uses it to validate the `userdata.json` file provided by the test runner. If valid, the contents of the `userdata.json` file are available in both the IDT context and in the test orchestrator context.

Configure the IDT test orchestrator

Starting in IDT v4.5.1, IDT includes a new *test orchestrator* component. The test orchestrator is an IDT component that controls the test suite execution flow, and generates the test report after IDT finishes running all tests. The test orchestrator determines test selection and the order in which tests are run based on user-defined rules.

If your test suite doesn't include a user-defined test orchestrator, IDT will generate a test orchestrator for you.

The default test orchestrator performs the following functions:

- Provides test runners with the ability to select and run specific test groups, instead of the entire test suite.
- If specific test groups are not selected, runs every test group in the test suite in a random order.
- Generates reports and prints a console summary that shows the test results for each test group and test case.

The test orchestrator replaces the IDT test orchestrator. We strongly recommend that you use the test orchestrator to develop your test suites instead of the IDT test orchestrator. The test orchestrator provides the following improved features:

- Uses a declarative format compared to the imperative format that the IDT state machine uses. This allows you to specify which tests you want to run and when you want to run them.
- Manages specific group handling, report generation, error handling, and result tracking so that you aren't required to manually manage these actions.
- Uses the YAML format, which supports comments by default.
- Requires 80 percent less disk space than the test orchestrator to define the same workflow.
- Adds pre-test validation to verify that your workflow definition doesn't contain incorrect test IDs or circular dependencies.

Test orchestrator format

You can use the following template to configure your own *<custom-test-suite-folder>*/`suite/test_orchestrator.yaml` file:

```
Aliases:
  string: context-expression

ConditionalTests:
  - Condition: context-expression
    Tests:
      - test-descriptor

Order:
  - - group-descriptor
    - group-descriptor

Features:
  - Name: feature-name
    Value: support-description
    Condition: context-expression
    Tests:
        - test-descriptor
    OneOfTests:
        - test-descriptor
    IsRequired: boolean
```

All fields that contain values are required as described here:

`Aliases`

Optional. User-defined strings that map to context expressions. Aliases allow you to generate friendly names to identify context expressions in your test orchestrator configuration. This is especially useful if you're creating complex context expressions or expressions that you use in multiple places.

You can use context expressions to store context queries that allow you to access data from other IDT configurations. For more information, see Access data in the context.

Example

```
Aliases:
    FizzChosen: "'{{$pool.features[?(@.name == 'Fizz')].value[0]}}' == 'yes'"
    BuzzChosen: "'{{$pool.features[?(@.name == 'Buzz')].value[0]}}' == 'yes'"
    FizzBuzzChosen: "'{{$aliases.FizzChosen}}' && '{{$aliases.BuzzChosen}}'"
```

`ConditionalTests`

Optional. A list of conditions, and the corresponding test cases that are run when each condition is satisfied. Each condition can have multiple test cases; however, you can assign a given test case to only one condition.

By default, IDT runs any test case that isn't assigned to a condition in this list. If you don't specify this section, IDT runs all test groups in the test suite.

Each item in the `ConditionalTests` list includes the following parameters:

`Condition`

A context expression that evaluates to a Boolean value. If the evaluated value is true, IDT runs the test cases that are specified in the `Tests` parameter.

`Tests`

The list of test descriptors.

Each test descriptor uses the test group ID and one or more test case IDs to identify the individual tests to run from a specific test group. The test descriptor uses the following format:

```
GroupId: group-id
CaseIds: [test-id, test-id] # optional
```

Example

The following example uses generic context expressions that you can define as `Aliases`.

```
ConditionalTests:
    - Condition: "{{$aliases.Condition1}}"
      Tests:
          - GroupId: A
          - GroupId: B
    - Condition: "{{$aliases.Condition2}}"
      Tests:
          - GroupId: D
    - Condition: "{{$aliases.Condition1}} || {{$aliases.Condition2}}"
      Tests:
          - GroupId: C
```

Based on the defined conditions, IDT selects test groups as follows:

- If `Condition1` is true, IDT runs the tests in test groups A, B, and C.
- If `Condition2` is true, IDT runs the tests in test groups C and D.

`Order`

Optional. The order in which to run tests. You specify the test order at the test group level. If you don't specify this section, IDT runs all applicable test groups in a random order. The value of `Order` is a list of group descriptor lists. Any test group that you don't list in `Order`, can be run in parallel with any other listed test group.

Each group descriptor list contains one of more group descriptors, and identifies the order in which to run the groups that are specified in each descriptor. You can use the following formats to define individual group descriptors:

- *`group-id`*—The group ID of an existing test group.
- [*`group-id`*, *`group-id`*]—List of test groups that can be run in any order relative to each other.
- `"*"`—Wildcard. This is equivalent to the list of all test groups that are not already specified in the current group descriptor list.

The value for `Order` must also meet the following requirements:

- Test group IDs that you specify in a group descriptor must exist in your test suite.
- Each group descriptor list must include at least one test group.
- Each group descriptor list must contain unique group IDs. You cannot repeat a test group ID within individual group descriptors.
- A group descriptor list can have at most one wildcard group descriptor. The wildcard group descriptor must be the first or the last item in the list.

Examples

For a test suite that contains test groups A, B, C, D, and E, the following list of examples shows different ways to specify that IDT should first run test group A, then run test group B, and then run test groups C, D, and E in any order.

- ```
  Order:
      - - A
        - B
        - [C, D, E]
  ```
- ```
  Order:
      - - A
        - B
        - "*"
  ```
- ```
  Order:
      - - A
        - B

      - - B
        - C

      - - B
        - D

      - - B
        - E
  ```

`Features`

Optional. The list of product features that you want IDT to add to the `awsiotdevicetester_report.xml` file. If you don't specify this section, IDT won't add any product features to the report.
  ```

A product feature is user-defined information about specific criteria that a device might meet. For example, the MQTT product feature can designate that the device publishes MQTT messages properly. In `awsiotdevicetester_report.xml`, product features are set as `supported`, `not-supported`, or a custom user-defined value, based on whether specified tests passed.

Each item in the `Features` list consists of the following parameters:

`Name`

The name of the feature.

`Value`

Optional. The custom value that you want to use in the report instead of `supported`. If this value is not specified, then based IDT sets the feature value to `supported` or `not-supported` based on test results. If you test the same feature with different conditions, you can use a custom value for each instance of that feature in the `Features` list, and IDT concatenates the feature values for supported conditions. For more information, see

`Condition`

A context expression that evaluates to a Boolean value. If the evaluated value is true, IDT adds the feature to the test report after it finishes running the test suite. If the evaluated value is false, the test is not included in the report.

`Tests`

Optional. The list of test descriptors. All of the tests that are specified in this list must pass for the feature to be supported.

Each test descriptor in this list uses the test group ID and one or more test case IDs to identify the individual tests to run from a specific test group. The test descriptor uses the following format:

```
GroupId: group-id
CaseIds: [test-id, test-id] # optional
```

You must specify either `Tests` or `OneOfTests` for each feature in the `Features` list.

`OneOfTests`

Optional. The list of test descriptors. At least one of the tests that are specified in this list must pass for the feature to be supported.

Each test descriptor in this list uses the test group ID and one or more test case IDs to identify the individual tests to run from a specific test group. The test descriptor uses the following format:

```
GroupId: group-id
CaseIds: [test-id, test-id] # optional
```

You must specify either `Tests` or `OneOfTests` for each feature in the `Features` list.

`IsRequired`

The boolean value that defines whether the feature is required in the test report. The default value is `false`.

Example

Test orchestrator context

The test orchestrator context is a read-only JSON document that contains data that is available to the test orchestrator during execution. The test orchestrator context is accessible only from the test orchestrator, and contains information that determines the test flow. For example, you can use information configured by test runners in the `userdata.json` file to determine whether a specific test is required to run.

The test orchestrator context uses the following format:

```
{
    "pool": {
        <device-json-pool-element>
    },
    "userData": {
        <userdata-json-content>
    },
    "config": {
        <config-json-content>
    }
}
```

`pool`

Information about the device pool selected for the test run. For a selected device pool, this information is retrieved from the corresponding top-level device pool array element defined in the `device.json` file.

`userData`

Information in the `userdata.json` file.

`config`

Information in the `config.json` file.

You can query the context using JSONPath notation. The syntax for JSONPath queries in state definitions is `{{query}}`. When you access data from the test orchestrator context, make sure that each value evaluates to a string, a number, or a Boolean.

For more information about using JSONPath notation to access data from the context, see Use the IDT context.

Configure the IDT state machine

Important

Starting in IDT v4.5.1, this state machine is deprecated. We strongly recommend that you use the new test orchestrator. For more information, see Configure the IDT test orchestrator.

A state machine is a construct that controls the test suite execution flow. It determines the starting state of a test suite, manages state transitions based on user-defined rules, and continues to transition through those states until it reaches the end state.

If your test suite doesn't include a user-defined state machine, IDT will generate a state machine for you. The default state machine performs the following functions:

- Provides test runners with the ability to select and run specific test groups, instead of the entire test suite.
- If specific test groups are not selected, runs every test group in the test suite in a random order.

- Generates reports and prints a console summary that shows the test results for each test group and test case.

The state machine for an IDT test suite must meet the following criteria:

- Each state corresponds to an action for IDT to take, such as to run a test group or product a report file.
- Transitioning to a state executes the action associated with the state.
- Each state defines the transition rule for the next state.
- The end state must be either `Succeed` or `Fail`.

State machine format

You can use the following template to configure your own *<custom-test-suite-folder>*/`suite/state_machine.json` file:

```
{
  "Comment": "<description>",
  "StartAt": "<state-name>",
  "States": {
    "<state-name>": {
      "Type": "<state-type>",
      // Additional state configuration
    }

    // Required states
    "Succeed": {
      "Type": "Succeed"
    },
    "Fail": {
      "Type": "Fail"
    }
  }
}
```

All fields that contain values are required as described here:

`Comment`

A description of the state machine.

`StartAt`

The name of the state at which IDT starts running the test suite. The value of `StartAt` must be set to one of the states listed in the `States` object.

`States`

An object that maps user-defined state names to valid IDT states. Each States.*`state-name`* object contains the definition of a valid state mapped to the *`state-name`*.

The `States` object must include the `Succeed` and `Fail` states. For information about valid states, see Valid states and state definitions.

Valid states and state definitions

This section describes the state definitions of all of the valid states that can be used in the IDT state machine. Some of the following states support configurations at the test case level. However, we recommend that you configure state transition rules at the test group level instead of the test case level unless absolutely necessary.

State definitions

- RunTask
- Choice
- Parallel
- AddProductFeatures
- Report
- LogMessage
- SelectGroup
- Fail
- Succeed

RunTask

The `RunTask` state runs test cases from a test group defined in the test suite.

```
{
```

```
    "Type": "RunTask",
    "Next": "<state-name>",
    "TestGroup": "<group-id>",
    "TestCases": [
        "<test-id>"
    ],
    "ResultVar": "<result-name>"
}
```

All fields that contain values are required as described here:

`Next`

The name of the state to transition to after executing the actions in the current state.

`TestGroup`

Optional. The ID of the test group to run. If this value is not specified, then IDT runs the test group that the test runner selects.

`TestCases`

Optional. An array of test case IDs from the group specified in `TestGroup`. Based on the values of `TestGroup` and `TestCases`, IDT determines the test execution behavior as follows:

- When both `TestGroup` and `TestCases` are specified, IDT runs the specified test cases from the test group.
- When `TestCases` are specified but `TestGroup` is not specified, IDT runs the specified test cases.
- When `TestGroup` is specified, but `TestCases` is not specified, IDT runs all of the test cases within the specified test group.
- When neither `TestGroup` or `TestCases` is specified, IDT runs all test cases from the test group that the test runner selects from the IDT CLI. To enable group selection for test runners, you must include both `RunTask` and `Choice` states in your `state_machine.json` file. For an example of how this works, see Example state machine: Run user-selected test groups.

  For more information about enabling IDT CLI commands for test runners, see the section called “Enable IDT CLI commands”.

`ResultVar`

The name of the context variable to set with the results of the test run. Do not specify this value if you did not specify a value for `TestGroup`. IDT sets the value of the variable that you define in `ResultVar` to `true` or `false` based on the following:

- If the variable name is of the form *`text`*`_`*`text`*`_passed`, then the value is set to whether all tests in the first test group passed or were skipped.
- In all other cases, the value is set to whether all tests in all test groups passed or were skipped.

Typically, you will use `RunTask` state to specify a test group ID without specifying individual test case IDs, so that IDT will run all of the test cases in the specified test group. All test cases that are run by this state run in parallel, in a random order. However, if all of the test cases require a device to run, and only a single device is available, then the test cases will run sequentially instead.

Error handling

If any of the specified test groups or test case IDs are not valid, then this state issues the `RunTaskError` execution error. If the state encounters an execution error, then it also sets the `hasExecutionError` variable in the state machine context to `true`.

Choice

The `Choice` state lets you dynamically set the next state to transition to based on user-defined conditions.

```
{
    "Type": "Choice",
    "Default": "<state-name>",
    "FallthroughOnError": true | false,
    "Choices": [
        {
            "Expression": "<expression>",
            "Next": "<state-name>"
        }
    ]
}
```

All fields that contain values are required as described here:

`Default`

The default state to transition to if none of the expressions defined in `Choices` can be evaluated to `true`.

`FallthroughOnError`

Optional. Specifies the behavior when the state encounters an error in evaluating expressions. Set to `true` if you want to skip an expression if the evaluation results in an error. If no expressions match, then the state machine transitions to the `Default` state. If the `FallthroughOnError` value is not specified, it defaults to `false`.

`Choices`

An array of expressions and states to determine which state to transition to after executing the actions in the current state.

`Choices.Expression`

An expression string that evaluates to a boolean value. If the expression evaluates to `true`, then the state machine transitions to the state defined in `Choices.Next`. Expression strings retrieve values from the state machine context and then perform operations on them to arrive at a boolean value. For information about accesing the state machine context, see State machine context.

`Choices.Next`

The name of the state to transition to if the expression defined in `Choices.Expression` evaluates to `true`.

Error handling

The `Choice` state can require error handling in the following cases:

- Some variables in the choice expressions don't exist in the state machine context.
- The result of an expression is not a boolean value.
- The result of a JSON lookup is not a string, number, or boolean.

You cannot use a `Catch` block to handle errors in this state. If you want to stop executing the state machine when it encounters an error, you must set `FallthroughOnError` to `false`. However, we

recommend that you set `FallthroughOnError` to `true`, and depending on your use case, do one of the following:

- If a variable you are accessing is expected to not exist in some cases, then use the value of `Default` and additional `Choices` blocks to specify the next state.
- If a variable that you are accessing should always exist, then set the `Default` state to `Fail`.

Parallel

The `Parallel` state lets you define and run new state machines in parallel with each other.

```
{
    "Type": "Parallel",
    "Next": "<state-name>",
    "Branches": [
        <state-machine-definition>
    ]
}
```

All fields that contain values are required as described here:

`Next`

The name of the state to transition to after executing the actions in the current state.

`Branches`

An array of state machine definitions to run. Each state machine definition must contain its own `StartAt`, `Succeed`, and `Fail` states. The state machine definitions in this array cannot reference states outside of their own definition.

> **Note**
>
> Because each branch state machine shares the same state machine context, setting variables in one branch and then reading those variables from another branch might result in unexpected behavior.

The `Parallel` state moves to the next state only after it runs all of the branch state machines. Each state that requires a device will wait to run until the device is available. If multiple devices

are available, this state runs test cases from multiple groups in parallel. If enough devices are not available, then test cases will run sequentially. Because test cases are run in a random order when they run in parallel, different devices might be used to run tests from the same test group.

Error handling

Make sure that both the branch state machine and the parent state machine transition to the `Fail` state to handle execution errors.

Because branch state machines do not transmit execution errors to the parent state machine, you cannot use a `Catch` block to handle execution errors in branch state machines. Instead, use the `hasExecutionErrors` value in the shared state machine context. For an example of how this works, see Example state machine: Run two test groups in parallel.

AddProductFeatures

The `AddProductFeatures` state lets you add product features to the `awsiotdevicetester_report.xml` file generated by IDT.

A product feature is user-defined information about specific criteria that a device might meet. For example, the `MQTT` product feature can designate that the device publishes MQTT messages properly. In the report, product features are set as `supported`, `not-supported`, or a custom value, based on whether specified tests passed.

> **Note**
>
> The `AddProductFeatures` state does not generate reports by itself. This state must transition to the `Report` state to generate reports.

```
{
    "Type": "Parallel",
    "Next": "<state-name>",
    "Features": [
        {
            "Feature": "<feature-name>",
            "Groups": [
                "<group-id>"
            ],
```

```
            "OneOfGroups": [
                "<group-id>"
            ],
            "TestCases": [
                "<test-id>"
            ],
            "IsRequired": true | false,
            "ExecutionMethods": [
                "<execution-method>"
            ]
        }
    ]
}
```

All fields that contain values are required as described here:

`Next`

The name of the state to transition to after executing the actions in the current state.

`Features`

An array of product features to show in the `awsiotdevicetester_report.xml` file.

`Feature`

The name of the feature

`FeatureValue`

Optional. The custom value to use in the report instead of `supported`. If this value is not specified, then based on test results, the feature value is set to `supported` or `not-supported`.

If you use a custom value for `FeatureValue`, you can test the same feature with different conditions, and IDT concatenates the feature values for the supported conditions. For example, the following excerpt shows the `MyFeature` feature with two separate feature values:

```
...
{
    "Feature": "MyFeature",
    "FeatureValue": "first-feature-supported",
    "Groups": ["first-feature-group"]
```

```
},
{
    "Feature": "MyFeature",
    "FeatureValue": "second-feature-supported",
    "Groups": ["second-feature-group"]
},
...
```

If both test groups pass, then the feature value is set to `first-feature-supported, second-feature-supported`.

`Groups`

Optional. An array of test group IDs. All tests within each specified test group must pass for the feature to be supported.

`OneOfGroups`

Optional. An array of test group IDs. All tests within at least one of the specified test groups must pass for the feature to be supported.

`TestCases`

Optional. An array of test case IDs. If you specify this value, then the following apply:

- All of the specified test cases must pass for the feature to be supported.
- `Groups` must contain only one test group ID.
- `OneOfGroups` must not be specified.

`IsRequired`

Optional. Set to `false` to mark this feature as an optional feature in the report. The default value is `true`.

`ExecutionMethods`

Optional. An array of execution methods that match the `protocol` value specified in the `device.json` file. If this value is specified, then test runners must specify a `protocol` value that matches one of the values in this array to include the feature in the report. If this value is not specified, then the feature will always be included in the report.

To use the `AddProductFeatures` state, you must set the value of `ResultVar` in the `RunTask` state to one of the following values:

- If you specified individual test case IDs, then set `ResultVar` to *`group-id_test-id`*`_passed`.
- If you did not specify individual test case IDs, then set `ResultVar` to *`group-id`*`_passed`.

The `AddProductFeatures` state checks for test results in the following manner:

- If you did not specify any test case IDs, then the result for each test group is determined from the value of the *`group-id`*`_passed` variable in the state machine context.
- If you did specify test case IDs, then the result for each of the tests is determined from the value of the *`group-id_test-id`*`_passed` variable in the state machine context.

Error handling

If a group ID provided in this state is not a valid group ID, then this state results in the `AddProductFeaturesError` execution error. If the state encounters an execution error, then it also sets the `hasExecutionErrors` variable in the state machine context to `true`.

Report

The `Report` state generates the *`suite-name`*`_Report.xml` and `awsiotdevicetester_report.xml` files. This state also streams the report to the console.

```
{
    "Type": "Report",
    "Next": "<state-name>"
}
```

All fields that contain values are required as described here:

`Next`

The name of the state to transition to after executing the actions in the current state.

You should always transition to the `Report` state towards the end of the test execution flow so that test runners can view test results. Typically, the next state after this state is `Succeed`.

Error handling

If this state encounters issues with generating the reports, then it issues the `ReportError` execution error.

LogMessage

The `LogMessage` state generates the `test_manager.log` file and streams the log message to the console.

```
{
    "Type": "LogMessage",
    "Next": "<state-name>"
    "Level": "info | warn | error"
    "Message": "<message>"
}
```

All fields that contain values are required as described here:

`Next`

The name of the state to transition to after executing the actions in the current state.

`Level`

The error level at which to create the log message. If you specify a level that is not valid, this state generates an error message and discards it.

`Message`

The message to log.

SelectGroup

The `SelectGroup` state updates the state machine context to indicate which groups are selected. The values set by this state are used by any subsequent `Choice` states.

```
{
    "Type": "SelectGroup",
    "Next": "<state-name>"
    "TestGroups": [
        <group-id>"
    ]
}
```

All fields that contain values are required as described here:

`Next`

The name of the state to transition to after executing the actions in the current state.

`TestGroups`

An array of test groups that will be marked as selected. For each test group ID in this array, the *`group-id`*`_selected` variable is set to `true` in the context. Make sure that you provide valid test group IDs because IDT does not validate whether the specified groups exist.

Fail

The `Fail` state indicates that the state machine did not execute correctly. This is an end state for the state machine, and each state machine definition must include this state.

```
{
    "Type": "Fail"
}
```

Succeed

The `Succeed` state indicates that the state machine executed correctly. This is an end state for the state machine, and each state machine definition must include this state.

```
{
    "Type": "Succeed"
}
```

State machine context

The state machine context is a read-only JSON document that contains data that is available to the state machine during execution. The state machine context is accessible only from the state machine, and contains information that determines the test flow. For example, you can use information configured by test runners in the `userdata.json` file to determine whether a specific test is required to run.

The state machine context uses the following format:

```
{
    "pool": {
        <device-json-pool-element>
```

```
    },
    "userData": {
        <userdata-json-content>
    },
    "config": {
        <config-json-content>
    },
    "suiteFailed": true | false,
    "specificTestGroups": [
        "<group-id>"
    ],
    "specificTestCases": [
        "<test-id>"
    ],
    "hasExecutionErrors": true
}
```

`pool`

Information about the device pool selected for the test run. For a selected device pool, this information is retrieved from the corresponding top-level device pool array element defined in the `device.json` file.

`userData`

Information in the `userdata.json` file.

`config`

Information pin the `config.json` file.

`suiteFailed`

The value is set to `false` when the state machine starts. If a test group fails in a `RunTask` state, then this value is set to `true` for the remaining duration of the state machine execution.

`specificTestGroups`

If the test runner selects specific test groups to run instead of the entire test suite, this key is created and contains the list of specific test group IDs.

`specificTestCases`

If the test runner selects specific test cases to run instead of the entire test suite, this key is created and contains the list of specific test case IDs.

`hasExecutionErrors`

Does not exit when the state machine starts. If any state encounters an execution errors, this variable is created and set to `true` for the remaining duration of the state machine execution.

You can query the context using JSONPath notation. The syntax for JSONPath queries in state definitions is `{{$.query}}`. You can use JSONPath queries as placeholder strings within some states. IDT replaces the placeholder strings with the value of the evaluated JSONPath query from the context. You can use placeholders for the following values:

- The `TestCases` value in `RunTask` states.
- The `Expression` value `Choice` state.

When you access data from the state machine context, make sure the following conditions are met:

- Your JSON paths must begin with `$.`
- Each value must evaluate to a string, a number, or a boolean.

For more information about using JSONPath notation to access data from the context, see Use the IDT context.

Execution errors

Execution errors are errors in the state machine definition that the state machine encounters when executing a state. IDT logs information about each error in the `test_manager.log` file and streams the log message to the console.

You can use the following methods to handle execution errors:

- Add a `Catch` block in the state definition.
- Check the value of the `hasExecutionErrors` value in the state machine context.

Catch

To use `Catch`, add the following to your state definition:

```
"Catch": [
    {
```

```
        "ErrorEquals": [
            "<error-type>"
        ]
        "Next": "<state-name>"
    }
]
```

All fields that contain values are required as described here:

`Catch.ErrorEquals`

An array of the error types to catch. If an execution error matches one of the specified values, then the state machine transitions to the state specified in `Catch.Next`. See each state definition for information about the type of error it produces.

`Catch.Next`

The next state to transition to if the current state encounters an execution error that matches one of the values specified in `Catch.ErrorEquals` .

Catch blocks are handled sequentially until one matches. If the no errors match the ones listed in the Catch blocks, then the state machines continues to execute. Because execution errors are a result of incorrect state definitions, we recommend that you transition to the Fail state when a state encounters an execution error.

hasExecutionError

When some states encounter execution errors, in addition to issuing the error, they also set the `hasExecutionError` value to `true` in the state machine context. You can use this value to detect when an error occurs, and then use a `Choice` state to transition the state machine to the `Fail` state.

This method has the following characteristics.

- The state machine does not start with any value assigned to `hasExecutionError`, and this value is not available until a particular state sets it. This means that you must explicitly set the `FallthroughOnError` to `false` for the `Choice` states that access this value to prevent the state machine from stopping if no execution errors occur.
- Once it is set to `true`, `hasExecutionError` is never set to false or removed from the context. This means that this value is useful only the first time that it is set to `true`, and for all subsequent states, it does not provide a meaningful value.

- The `hasExecutionError` value is shared with all branch state machines in the `Parallel` state, which can result in unexpected results depending on the order in which it is accessed.

Because of these characteristics, we do not recommend that you use this method if you can use a Catch block instead.

Example state machines

This section provides some example state machine configurations.

Examples

- Example state machine: Run a single test group
- Example state machine: Run user-selected test groups
- Example state machine: Run a single test group with product features
- Example state machine: Run two test groups in parallel

Example state machine: Run a single test group

This state machine:

- Runs the test group with id `GroupA`, which must be present in the suite in a `group.json` file.
- Checks for execution errors and transitions to `Fail` if any are found.
- Generates a report and transitions to `Succeed` if there are no errors, and `Fail` otherwise.

```
{
    "Comment": "Runs a single group and then generates a report.",
    "StartAt": "RunGroupA",
    "States": {
        "RunGroupA": {
            "Type": "RunTask",
            "Next": "Report",
            "TestGroup": "GroupA",
            "Catch": [
                {
                    "ErrorEquals": [
                        "RunTaskError"
                    ],
                    "Next": "Fail"
```

```
                }
            ]
        },
        "Report": {
            "Type": "Report",
            "Next": "Succeed",
            "Catch": [
                {
                    "ErrorEquals": [
                        "ReportError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
        "Succeed": {
            "Type": "Succeed"
        },
        "Fail": {
            "Type": "Fail"
        }
    }
}
```

Example state machine: Run user-selected test groups

This state machine:

- Checks if the test runner selected specific test groups. The state machine does not check for specific test cases because test runners cannot select test cases without also selecting a test group.
- If test groups are selected:
  - Runs the test cases within the selected test groups. To do so, the state machine does not explicitly specify any test groups or test cases in the `RunTask` state.
  - Generates a report after running all tests and exits.
- If test groups are not selected:
  - Runs tests in test group `GroupA`.
  - Generates reports and exits.

```
{
    "Comment": "Runs specific groups if the test runner chose to do that, otherwise
 runs GroupA.",
    "StartAt": "SpecificGroupsCheck",
    "States": {
        "SpecificGroupsCheck": {
            "Type": "Choice",
            "Default": "RunGroupA",
            "FallthroughOnError": true,
            "Choices": [
                {
                    "Expression": "{{$.specificTestGroups[0]}} != ''",
                    "Next": "RunSpecificGroups"
                }
            ]
        },
        "RunSpecificGroups": {
            "Type": "RunTask",
            "Next": "Report",
            "Catch": [
                {
                    "ErrorEquals": [
                        "RunTaskError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
        "RunGroupA": {
            "Type": "RunTask",
            "Next": "Report",
            "TestGroup": "GroupA",
            "Catch": [
                {
                    "ErrorEquals": [
                        "RunTaskError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
        "Report": {
            "Type": "Report",
```

```
            "Next": "Succeed",
            "Catch": [
                {
                    "ErrorEquals": [
                        "ReportError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
        "Succeed": {
            "Type": "Succeed"
        },
        "Fail": {
            "Type": "Fail"
        }
    }
}
```

Example state machine: Run a single test group with product features

This state machine:

- Runs the test group `GroupA`.
- Checks for execution errors and transitions to `Fail` if any are found.
- Adds the `FeatureThatDependsOnGroupA` feature to the `awsiotdevicetester_report.xml` file:
  - If `GroupA` passes, the feature is set to `supported`.
  - The feature is not marked optional in the report.
- Generates a report and transitions to `Succeed` if there are no errors, and `Fail` otherwise

```
{
    "Comment": "Runs GroupA and adds product features based on GroupA",
    "StartAt": "RunGroupA",
    "States": {
        "RunGroupA": {
            "Type": "RunTask",
            "Next": "AddProductFeatures",
            "TestGroup": "GroupA",
            "ResultVar": "GroupA_passed",
```

```
            "Catch": [
                {
                    "ErrorEquals": [
                        "RunTaskError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
        "AddProductFeatures": {
            "Type": "AddProductFeatures",
            "Next": "Report",
            "Features": [
                {
                    "Feature": "FeatureThatDependsOnGroupA",
                    "Groups": [
                        "GroupA"
                    ],
                    "IsRequired": true
                }
            ]
        },
        "Report": {
            "Type": "Report",
            "Next": "Succeed",
            "Catch": [
                {
                    "ErrorEquals": [
                        "ReportError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
        "Succeed": {
            "Type": "Succeed"
        },
        "Fail": {
            "Type": "Fail"
        }
    }
}
```

Example state machine: Run two test groups in parallel

This state machine:

- Runs the `GroupA` and `GroupB` test groups in parallel. The `ResultVar` variables stored in the context by the `RunTask` states in the branch state machines by are available to the `AddProductFeatures` state.
- Checks for execution errors and transitions to `Fail` if any are found. This state machine does not use a `Catch` block because that method does not detect execution errors in branch state machines.
- Adds features to the `awsiotdevicetester_report.xml` file based on the groups that pass
  - If `GroupA` passes, the feature is set to `supported`.
  - The feature is not marked optional in the report.
- Generates a report and transitions to `Succeed` if there are no errors, and `Fail` otherwise

If two devices are configured in the device pool, both `GroupA` and `GroupB` can run at the same time. However, if either `GroupA` or `GroupB` has multiple tests in it, then both devices may be allocated to those tests. If only one device is configured, the test groups will run sequentially.

```
{
    "Comment": "Runs GroupA and GroupB in parallel",
    "StartAt": "RunGroupAAndB",
    "States": {
        "RunGroupAAndB": {
            "Type": "Parallel",
            "Next": "CheckForErrors",
            "Branches": [
                {
                    "Comment": "Run GroupA state machine",
                    "StartAt": "RunGroupA",
                    "States": {
                        "RunGroupA": {
                            "Type": "RunTask",
                            "Next": "Succeed",
                            "TestGroup": "GroupA",
                            "ResultVar": "GroupA_passed",
                            "Catch": [
                                {
                                    "ErrorEquals": [
```

```
                                    "RunTaskError"
                                ],
                                "Next": "Fail"
                            }
                        ]
                    },
                    "Succeed": {
                        "Type": "Succeed"
                    },
                    "Fail": {
                        "Type": "Fail"
                    }
                }
            },
            {
                "Comment": "Run GroupB state machine",
                "StartAt": "RunGroupB",
                "States": {
                    "RunGroupA": {
                        "Type": "RunTask",
                        "Next": "Succeed",
                        "TestGroup": "GroupB",
                        "ResultVar": "GroupB_passed",
                        "Catch": [
                            {
                                "ErrorEquals": [
                                    "RunTaskError"
                                ],
                                "Next": "Fail"
                            }
                        ]
                    },
                    "Succeed": {
                        "Type": "Succeed"
                    },
                    "Fail": {
                        "Type": "Fail"
                    }
                }
            }
        ]
    },
    "CheckForErrors": {
        "Type": "Choice",
```

```
            "Default": "AddProductFeatures",
            "FallthroughOnError": true,
            "Choices": [
                {
                    "Expression": "{{$.hasExecutionErrors}} == true",
                    "Next": "Fail"
                }
            ]
        },
        "AddProductFeatures": {
            "Type": "AddProductFeatures",
            "Next": "Report",
            "Features": [
                {
                    "Feature": "FeatureThatDependsOnGroupA",
                    "Groups": [
                        "GroupA"
                    ],
                    "IsRequired": true
                },
                {
                    "Feature": "FeatureThatDependsOnGroupB",
                    "Groups": [
                        "GroupB"
                    ],
                    "IsRequired": true
                }
            ]
        },
        "Report": {
            "Type": "Report",
            "Next": "Succeed",
            "Catch": [
                {
                    "ErrorEquals": [
                        "ReportError"
                    ],
                    "Next": "Fail"
                }
            ]
        },
        "Succeed": {
            "Type": "Succeed"
        },
```

```
        "Fail": {
            "Type": "Fail"
        }
    }
}
```

Create IDT test case executables

You can create and place test case executables in a test suite folder in the following ways:

- For test suites that use arguments or environment variables from the `test.json` files to determine which tests to run, you can create a single test case executable for the entire test suite, or a test executable for each test group in the test suite.
- For a test suite where you want to run specific tests based on specified commands, you create one test case executable for each test case in the test suite.

As a test writer, you can determine which approach is appropriate for your use case and structure your test case executable accordingly. Make sure that you provide the correct test case executable path in each `test.json` file, and that the specified executable runs correctly.

When all devices are ready for a test case to run, IDT reads the following files:

- The `test.json` for the selected test case determines the processes to start and the environment variables to set.
- The `suite.json` for the test suite determines the environment variables to set.

IDT starts the required test executable process based on the commands and arguments specified in the `test.json` file, and passes the required environment variables to the process.

Use the IDT Client SDK

The IDT Client SDKs let you simplify how you write test logic in your test executable with API commands that you can use interact with IDT and your devices under test. IDT currently provides the following SDKs:

- IDT Client SDK for Python
- IDT Client SDK for Go
- IDT Client SDK for Java

These SDKs are located in the *<device-tester-extract-location>*/sdks folder. When you create a new test case executable, you must copy the SDK that you want to use to the folder that contains your test case executable and reference the SDK in your code. This section provides a brief description of the available API commands that you can use in your test case executables.

In this section

- Device interaction
- IDT interaction
- Host interaction

Device interaction

The following commands enable you to communicate with the device under test without having to implement any additional device interaction and connectivity management functions.

`ExecuteOnDevice`

Allows test suites to run shell commands on a device that support SSH or Docker shell connections.

`CopyToDevice`

Allows test suites to copy a local file from the host machine that runs IDT to a specified location on a device that supports SSH or Docker shell connections.

`ReadFromDevice`

Allows test suites to read from the serial port of devices that support UART connections.

> **Note**
>
> Because IDT does not manage direct connections to devices that are made using device access information from the context, we recommend using these device interaction API commands in your test case executables. However, if these commands do not meet your test case requirements, then you can retrieve device access information from the IDT context and use it to make a direct connection to the device from the test suite.
> To make a direct connection, retrieve the information in the `device.connectivity` and the `resource.devices.connectivity` fields for your device under test and for resource

devices, respectively. For more information about using the IDT context, see Use the IDT context.

IDT interaction

The following commands enable your test suites to communicate with IDT.

`PollForNotifications`

Allows test suites to check for notifications from IDT.

`GetContextValue` and `GetContextString`

Allows test suites to retrieve values from the IDT context. For more information, see Use the IDT context.

`SendResult`

Allows test suites to report test case results to IDT. This command must be called at the end of each test case in a test suite.

Host interaction

The following command enable your test suites to communicate with the host machine.

`PollForNotifications`

Allows test suites to check for notifications from IDT.

`GetContextValue` and `GetContextString`

Allows test suites to retrieve values from the IDT context. For more information, see Use the IDT context.

`ExecuteOnHost`

Allows test suites to run commands on the local machine and lets IDT manage the test case executable lifecycle.

Enable IDT CLI commands

The `run-suite` command IDT CLI provides several options that let test runner customize test execution. To allow test runners to use these options to run your custom test suite, you implement

support for the IDT CLI. If you do not implement support, test runners will still be able to run tests, but some CLI options will not function correctly. To provide an ideal customer experience, we recommend that you implement support for the following arguments for the `run-suite` command in the IDT CLI:

`timeout-multiplier`

Specifies a value greater than 1.0 that will be applied to all timeouts while running tests.

Test runners can use this argument to increase the timeout for the test cases that they want to run. When a test runner specifies this argument in their `run-suite` command, IDT uses it to calculate the value of the IDT_TEST_TIMEOUT environment variable and sets the `config.timeoutMultiplier` field in the IDT context. To support this argument, you must do the following:

- Instead of directly using the timeout value from the `test.json` file, read the IDT_TEST_TIMEOUT environment variable to obtain the correctly calculated timeout value.
- Retrieve the `config.timeoutMultiplier` value from the IDT context and apply it to long running timeouts.

For more information about exiting early because of timeout events, see Specify exit behavior.

`stop-on-first-failure`

Specifies that IDT should stop running all tests if it encounters a failure.

When a test runner specifies this argument in their `run-suite` command, IDT will stop running tests as soon as it encounters a failure. However, if test cases are running in parallel, then this can lead to unexpected results. To implement support, make sure that if IDT encounters this event, your test logic instructs all running test cases to stop, clean up temporary resources, and report a test result to IDT. For more information about exiting early on failures, see Specify exit behavior.

`group-id` and `test-id`

Specifies that IDT should run only the selected test groups or test cases.

Test runners can use these arguments with their `run-suite` command to specify the following test execution behavior:

- Run all tests inside the specified test groups.
- Run a selection of tests from within a specified test group.

To support these arguments, the test orchestrator for your test suite must include a specific set of `RunTask` and `Choice` states in your test orchestrator. If you are not using a custom state machine, then the default IDT test orchestrator includes the required states for you and you do not need to take additional action. However, if you are using a custom test orchestrator, then use Example state machine: Run user-selected test groups as a sample to add the required states in your test orchestrator.

For more information about IDT CLI commands, see Debug and run custom test suites.

Write event logs

While the test is running, you send data to `stdout` and `stderr` to write event logs and error messages to the console. For information about the format of console messages, see Console message format.

When the IDT finishes running the test suite, this information is also available in the `test_manager.log` file located in the *<devicetester-extract-location>*/`results/`*<execution-id>*`/logs` folder.

You can configure each test case to write the logs from its test run, including logs from the device under test, to the *<group-id>*_*<test-id>* file located in the *<device-tester-extract-location>*`/results/`*execution-id*`/logs` folder. To do this, retrieve the path to the log file from the IDT context with the `testData.logFilePath` query, create a file at that path, and write the content that you want to it. IDT automatically updates the path based on the test case that is running. If you choose not to create the log file for a test case, then no file is generated for that test case.

You can also set up your text executable to create additional log files as needed in the *<device-tester-extract-location>*`/logs` folder. We recommend that you specify unique prefixes for log file names so your files don't get overwritten.

Report results to IDT

IDT writes test results to the `awsiotdevicetester_report.xml` and the *suite-name*`_report.xml` files. These report files are located in *<device-tester-extract-location>*`/results/`*<execution-id>*`/`. Both reports capture the results from the test suite execution. For more information about the schemas that IDT uses for these reports, see Review IDT test results and logs

To populate the contents of the *suite-name*_report.xml file, you must use the SendResult command to report test results to IDT before the test execution finishes. If IDT cannot locate the results of a test, it issues an error for the test case. The following Python excerpt shows the commands to send a test result to IDT:

```
request-variable = SendResultRequest(TestResult(result))
client.send_result(request-variable)
```

If you do not report results through the API, IDT looks for test results in the test artifacts folder. The path to this folder is stored in the testData.testArtifactsPath filed in the IDT context. In this folder, IDT uses the first alphabetically sorted XML file it locates as the test result.

If your test logic produces JUnit XML results, you can write the test results to an XML file in the artifacts folder to directly provide the results to IDT instead of parsing the results and then using the API to submit them to IDT.

If you use this method, make sure that your test logic accurately summarizes the test results and format your result file in the same format as the *suite-name*_report.xml file. IDT does not perform any validation of the data that you provide, with the following exceptions:

- IDT ignores all properties of the testsuites tag. Instead, it calculates the tag properties from other reported test group results.
- At least one testsuite tag must exist within testsuites.

Because IDT uses the same artifacts folder for all test cases and does not delete result files between test runs, this method might also lead to erroneous reporting if IDT reads the incorrect file. We recommend that you use the same name for the generated XML results file across all test cases to overwrite the results for each test case and make sure that the correct results are available for IDT to use. Although you can use a mixed approach to reporting in your test suite, that is, use an XML result file for some test cases and submit results through the API for others, we do not recommend this approach.

Specify exit behavior

Configure your text executable to always exit with an exit code of 0, even if a test case reports a failure or an error result. Use non-zero exit codes only to indicate that a test case did not run or if the test case executable could not communicate any results to IDT. When IDT receives a non-zero exit code, it marks the test case has having encountered an error that prevented it from running.

IDT might request or expect a test case to stop running before it has finished in the following events. Use this information to configure your test case executable to detect each of these events from the test case:

Timeout

Occurs when a test case runs for longer than the timeout value specified in the `test.json` file. If the test runner used the `timeout-multiplier` argument to specify a timeout multiplier, then IDT calculates the timeout value with the multiplier.

To detect this event, use the IDT_TEST_TIMEOUT environment variable. When a test runner launches a test, IDT sets the value of the IDT_TEST_TIMEOUT environment variable to the calculated timeout value (in seconds) and passes the variable to the test case executable. You can read the variable value to set an appropriate timer.

Interrupt

Occurs when the test runner interrupts IDT. For example, by pressing **Ctrl+C**.

Because terminals propagate signals to all child processes, you can simply configure a signal handler in your test cases to detect interrupt signals.

Alternatively, you can periodically poll the API to check the value of the `CancellationRequested` boolean in the `PollForNotifications` API response. When IDT receives an interrupt signal, it sets the value of the `CancellationRequested` boolean to `true`.

Stop on first failure

Occurs when a test case that is running in parallel with the current test case fails and the test runner used the `stop-on-first-failure` argument to specify that IDT should stop when it encounters any failure.

To detect this event, you can periodically poll the API to check the value of the `CancellationRequested` boolean in the `PollForNotifications` API response. When IDT encounters a failure and is configured to stop on first failure, it sets the value of the `CancellationRequested` boolean to `true`.

When any of these events occur, IDT waits for 5 minutes for any currently running test cases to finish running. If all running test cases do not exit within 5 minutes, IDT forces each of their

processes to stop. If IDT has not received test results before the processes end, it will mark the test cases as having timed out. As a best practice, you should ensure that your test cases perform the following actions when they encounter one of the events:

1. Stop running normal test logic.
2. Clean up any temporary resources, such as test artifacts on the device under test.
3. Report a test result to IDT, such as a test failure or an error.
4. Exit.

Use the IDT context

When IDT runs a test suite, the test suite can access a set of data that can be used to determine how each test runs. This data is called the IDT context. For example, user data configuration provided by test runners in a `userdata.json` file is made available to test suites in the IDT context.

The IDT context can be considered a read-only JSON document. Test suites can retrieve data from and write data to the context using standard JSON data types like objects, arrays, numbers and so on.

Context schema

The IDT context uses the following format:

```
{
    "config": {
        <config-json-content>
        "timeoutMultiplier": timeout-multiplier
    },
    "device": {
        <device-json-device-element>
    },
    "devicePool": {
        <device-json-pool-element>
    },
    "resource": {
        "devices": [
            {
                <resource-json-device-element>
                "name": "<resource-name>"
```

```
            }
        ]
    },
    "testData": {
        "awsCredentials": {
            "awsAccessKeyId": "<access-key-id>",
            "awsSecretAccessKey": "<secret-access-key>",
            "awsSessionToken": "<session-token>"
        },
        "logFilePath": "/path/to/log/file"
    },
    "userData": {
        <userdata-json-content>
    }
}
```

`config`

Information from the `config.json` file. The `config` field also contains the following additional field:

`config.timeoutMultiplier`

The multiplier for the any timeout value used by the test suite. This value is specified by the test runner from the IDT CLI. The default value is 1.

`device`

Information about the device selected for the test run. This information is equivalent to the `devices` array element in the `device.json` file for the selected device.

`devicePool`

Information about the device pool selected for the test run. This information is equivalent to the top-level device pool array element defined in the `device.json` file for the selected device pool.

`resource`

Information about resource devices from the `resource.json` file.

`resource.devices`

This information is equivalent to the `devices` array defined in the `resource.json` file. Each `devices` element includes the following additional field:

`resource.device.name`

The name of the resource device. This value is set to the `requiredResource.name` value in the `test.json` file.

`testData.awsCredentials`

The AWS credentials used by the test to connect to the AWS cloud. This information is obtained from the `config.json` file.

`testData.logFilePath`

The path to the log file to which the test case writes log messages. The test suite creates this file if it doesn't exist.

`userData`

Information provided by the test runner in the `userdata.json` file.

Access data in the context

You can query the context using JSONPath notation from your JSON files and from your text executable with the `GetContextValue` and `GetContextString` APIs. The syntax for JSONPath strings to access the IDT context varies as follows:

- In `suite.json` and `test.json`, you use {{*query*}}. That is, do not use the root element `$.` to start your expression.
- In `test_orchestrator.yaml`, you use {{*query*}}.

  If you use the deprecated state machine, then in `state_machine.json`, you use {{`$.`*query*}}.
- In API commands, you use *query* or {{`$.`*query*}}, depending on the command. For more information, see the inline documentation in the SDKs.

The following table describes the operators in a typical JSONPath expression:

Operator	Description
$	The root element. Because the top-level context value for IDT is an object, you will typically use `$.` to start your queries.

Operator	Description
`.childName`	Accesses the child element with name `childName` from an object. If applied to an array, yields a new array with this operator applied to each element. The element name is case sensitive. For example, the query to access the `awsRegion` value in the `config` object is `$.config.awsRegion`.
`[start:end]`	Filters elements from an array, retrieving items beginning from the `start` index and going up to the `end` index, both inclusive.
`[index1, index2, ... , indexN]`	Filters elements from an array, retrieving items from only the specified indices.
`[?(expr)]`	Filters elements from an array using the `expr` expression. This expression must evaluate to a boolean value.

To create filter expressions, use the following syntax:

```
<jsonpath> | <value> operator <jsonpath> | <value>
```

In this syntax:

- `jsonpath` is a JSONPath that uses standard JSON syntax.
- `value` is any custom value that uses standard JSON syntax.
- `operator` is one of the following operators:
  - < (Less than)
  - <= (Less than or equal to)
  - == (Equal to)

    If the JSONPath or value in your expression is an array, boolean, or object value, then this is the only supported binary operator that you can use.

- >= (Greater than or equal to)
- > (Greater than)
- =~ (Regular expression match). To use this operator in a filter expression, the JSONPath or value on the left side of your expression must evaluate to a string and the right side must be a pattern value that follows the RE2 syntax.

You can use JSONPath queries in the form {{*query*}} as placeholder strings within the `args` and `environmentVariables` fields in `test.json` files and within the `environmentVariables` fields in `suite.json` files. IDT performs a context lookup and populates the fields with the evaluated value of the query. For example, in the `suite.json` file, you can use placeholder strings to specify environment variable values that change with each test case and IDT will populate the environment variables with the correct value for each test case. However, when you use placeholder strings in `test.json` and `suite.json` files, the following considerations apply for your queries:

- You must each occurrence of the `devicePool` key in your query in all lower case. That is, use `devicepool` instead.
- For arrays, you can use only arrays of strings. In addition, arrays use a non-standard `item1, item2,...,itemN` format. If the array contains only one element, then it is serialized as `item`, making it indistinguishable from a string field.
- You cannot use placeholders to retrieve objects from the context.

Because of these considerations, we recommend that whenever possible, you use the API to access the context in your test logic instead of placeholder strings in `test.json` and `suite.json` files. However, in some cases it might be more convenient to use JSONPath placeholders to retrieve single strings to set as environment variables.

Configure settings for test runners

To run custom test suites, test runners must configure their settings based on the test suite that they want to run. Settings are specified based on configuration file templates located in the *<device-tester-extract-location>*`/configs/` folder. If required, test runners must also set up AWS credentials that IDT will use to connect to the AWS cloud.

As a test writer, you will need to configure these files to debug your test suite. You must provide instructions to test runners so that they can configure the following settings as needed to run your test suites.

Configure device.json

The `device.json` file contains information about the devices that tests are run on (for example, IP address, login information, operating system, and CPU architecture).

Test runners can provide this information using the following template `device.json` file located in the *<device-tester-extract-location>*`/configs/` folder.

```
[
    {
        "id": "<pool-id>",
        "sku": "<pool-sku>",
        "features": [
            {
                "name": "<feature-name>",
                "value": "<feature-value>",
                "configs": [
                    {
                        "name": "<config-name>",
                        "value": "<config-value>"
                    }
                ],
            }
        ],
        "devices": [
            {
                "id": "<device-id>",
                "connectivity": {
                    "protocol": "ssh | uart | docker",
                    // ssh
                    "ip": "<ip-address>",
                    "port": <port-number>,
                    "auth": {
                        "method": "pki | password",
                        "credentials": {
                            "user": "<user-name>",
                            // pki
                            "privKeyPath": "/path/to/private/key",
```

```
                        // password
                        "password": "<password>",
                    }
                },

                // uart
                "serialPort": "<serial-port>",

                // docker
                "containerId": "<container-id>",
                "containerUser": "<container-user-name>",
            }
        }
    ]
}
]
```

All fields that contain values are required as described here:

`id`

A user-defined alphanumeric ID that uniquely identifies a collection of devices called a *device pool*. Devices that belong to a pool must have identical hardware. When you run a suite of tests, devices in the pool are used to parallelize the workload. Multiple devices are used to run different tests.

`sku`

An alphanumeric value that uniquely identifies the device under test. The SKU is used to track qualified devices.

> **Note**
>
> If you want to list your board in the AWS Partner Device Catalog, the SKU you specify here must match the SKU that you use in the listing process.

`features`

Optional. An array that contains the device's supported features. Device features are user-defined values that you configure in your test suite. You must provide your test runners with information about the feature names and values to include in the `device.json` file. For

example, if you want to test a device that functions as an MQTT server for other devices, then you can configure your test logic to validate specific supported levels for a feature named `MQTT_QOS`. Test runners provide this feature name and set the feature value to the QOS levels supported by their device. You can retrieve the provided information from the IDT context with the `devicePool.features` query, or from the test orchestrator context with the `pool.features` query.

`features.name`

The name of the feature.

`features.value`

The supported feature values.

`features.configs`

Configuration settings, if needed, for the feature.

`features.config.name`

The name of the configuration setting.

`features.config.value`

The supported setting values.

`devices`

An array of devices in the pool to be tested. At least one device is required.

`devices.id`

A user-defined unique identifier for the device being tested.

`connectivity.protocol`

The communication protocol used to communicate with this device. Each device in a pool must use the same protocol.

Currently, the only supported values are `ssh` and `uart` for physical devices, and `docker` for Docker containers.

`connectivity.ip`

The IP address of the device being tested.

This property applies only if `connectivity.protocol` is set to `ssh`.

`connectivity.port`

Optional. The port number to use for SSH connections.

The default value is 22.

This property applies only if `connectivity.protocol` is set to `ssh`.

`connectivity.auth`

Authentication information for the connection.

This property applies only if `connectivity.protocol` is set to `ssh`.

`connectivity.auth.method`

The authentication method used to access a device over the given connectivity protocol.

Supported values are:

- `pki`
- `password`

`connectivity.auth.credentials`

The credentials used for authentication.

`connectivity.auth.credentials.password`

The password used for signing in to the device being tested.

This value applies only if `connectivity.auth.method` is set to `password`.

`connectivity.auth.credentials.privKeyPath`

The full path to the private key used to sign in to the device under test.

This value applies only if `connectivity.auth.method` is set to `pki`.

`connectivity.auth.credentials.user`

The user name for signing in to the device being tested.

`connectivity.serialPort`

Optional. The serial port to which the device is connected.

This property applies only if `connectivity.protocol` is set to `uart`.

`connectivity.containerId`

The container ID or name of the Docker container being tested.

This property applies only if `connectivity.protocol` is set to `ssh`.

`connectivity.containerUser`

Optional. The name of the user to user inside the container. The default value is the user provided in the Dockerfile.

The default value is 22.

This property applies only if `connectivity.protocol` is set to `ssh`.

> **Note**
>
> To check if test runners configure the incorrect device connection for a test, you can retrieve `pool.Devices[0].Connectivity.Protocol` from the test orchestrator context and compare it to the expected value in a `Choice` state. If an incorrect protocol is used, then print a message using the `LogMessage` state and transition to the `Fail` state.
> Alternatively, you can use error handling code to report a test failure for incorrect device types.

(Optional) Configure userdata.json

The `userdata.json` file contains any additional information that is required by a test suite but is not specified in the `device.json` file. The format of this file depends on the `userdata_scheme.json` file that is defined in the test suite. If you are a test writer, make sure you provide this information to users who will run the test suites that you write.

(Optional) Configure resource.json

The `resource.json` file contains information about any devices that will be used as resource devices. Resource devices are devices that are required to test certain capabilities of a device under test. For example, to test a device's Bluetooth capability, you might use a resource device to test that your device can connect to it successfully. Resource devices are optional, and you can require

as many resources devices as you need. As a test writer, you use the test.json file to define the resource device features that are required for a test. Test runners then use the `resource.json` file to provide a pool of resource devices that have the required features. Make sure you provide this information to users who will run the test suites that you write.

Test runners can provide this information using the following template `resource.json` file located in the *<device-tester-extract-location>*`/configs/` folder.

```
[
    {
        "id": "<pool-id>",
        "features": [
            {
                "name": "<feature-name>",
                "version": "<feature-version>",
                "jobSlots": <job-slots>
            }
        ],
        "devices": [
            {
                "id": "<device-id>",
                "connectivity": {
                    "protocol": "ssh | uart | docker",
                    // ssh
                    "ip": "<ip-address>",
                    "port": <port-number>,
                    "auth": {
                        "method": "pki | password",
                        "credentials": {
                            "user": "<user-name>",
                            // pki
                            "privKeyPath": "/path/to/private/key",

                            // password
                            "password": "<password>",
                        }
                    },

                    // uart
                    "serialPort": "<serial-port>",

                    // docker
                    "containerId": "<container-id>",
```

```
                    "containerUser": "<container-user-name>",
                }
            }
        ]
    }
]
```

All fields that contain values are required as described here:

`id`

A user-defined alphanumeric ID that uniquely identifies a collection of devices called a *device pool*. Devices that belong to a pool must have identical hardware. When you run a suite of tests, devices in the pool are used to parallelize the workload. Multiple devices are used to run different tests.

`features`

Optional. An array that contains the device's supported features. The information required in this field is defined in the test.json files in the test suite and determines which tests to run and how to run those tests. If the test suite does not require any features, then this field is not required.

`features.name`

The name of the feature.

`features.version`

The feature version.

`features.jobSlots`

Setting to indicate how many tests can concurrently use the device. The default value is 1.

`devices`

An array of devices in the pool to be tested. At least one device is required.

`devices.id`

A user-defined unique identifier for the device being tested.

`connectivity.protocol`

The communication protocol used to communicate with this device. Each device in a pool must use the same protocol.

Currently, the only supported values are `ssh` and `uart` for physical devices, and `docker` for Docker containers.

`connectivity.ip`

The IP address of the device being tested.

This property applies only if `connectivity.protocol` is set to `ssh`.

`connectivity.port`

Optional. The port number to use for SSH connections.

The default value is 22.

This property applies only if `connectivity.protocol` is set to `ssh`.

`connectivity.auth`

Authentication information for the connection.

This property applies only if `connectivity.protocol` is set to `ssh`.

`connectivity.auth.method`

The authentication method used to access a device over the given connectivity protocol.

Supported values are:

- `pki`
- `password`

`connectivity.auth.credentials`

The credentials used for authentication.

`connectivity.auth.credentials.password`

The password used for signing in to the device being tested.

This value applies only if `connectivity.auth.method` is set to `password`.

`connectivity.auth.credentials.privKeyPath`

The full path to the private key used to sign in to the device under test.

This value applies only if `connectivity.auth.method` is set to `pki`.

`connectivity.auth.credentials.user`

The user name for signing in to the device being tested.

`connectivity.serialPort`

Optional. The serial port to which the device is connected.

This property applies only if `connectivity.protocol` is set to `uart`.

`connectivity.containerId`

The container ID or name of the Docker container being tested.

This property applies only if `connectivity.protocol` is set to `ssh`.

`connectivity.containerUser`

Optional. The name of the user to user inside the container. The default value is the user provided in the Dockerfile.

The default value is 22.

This property applies only if `connectivity.protocol` is set to `ssh`.

(Optional) Configure config.json

The `config.json` file contains configuration information for IDT. Typically, test runners will not need to modify this file except to provide their AWS user credentials for IDT, and optionally, an AWS region. If AWS credentials with required permissions are provided AWS IoT Device Tester collects and submits usage metrics to AWS. This is an opt-in feature and is used to improve IDT functionality. For more information, see IDT usage metrics.

Test runners can configure their AWS credentials in one of the following ways:

- **Credentials file**

  IDT uses the same credentials file as the AWS CLI. For more information, see Configuration and credential files.

  The location of the credentials file varies, depending on the operating system you are using:

- macOS, Linux: `~/.aws/credentials`
- Windows: `C:\Users\UserName\.aws\credentials`

- **Environment variables**

  Environment variables are variables maintained by the operating system and used by system commands. Variables defined during an SSH session are not available after that session is closed. IDT can use the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables to store AWS credentials

  To set these variables on Linux, macOS, or Unix, use **export**:

  ```
  export AWS_ACCESS_KEY_ID=<your_access_key_id>
  export AWS_SECRET_ACCESS_KEY=<your_secret_access_key>
  ```

  To set these variables on Windows, use **set**:

  ```
  set AWS_ACCESS_KEY_ID=<your_access_key_id>
  set AWS_SECRET_ACCESS_KEY=<your_secret_access_key>
  ```

To configure AWS credentials for IDT, test runners edit the `auth` section in the `config.json` file located in the *<device-tester-extract-location>*`/configs/` folder.

```
{
    "log": {
        "location": "logs"
    },
    "configFiles": {
        "root": "configs",
        "device": "configs/device.json"
    },
    "testPath": "tests",
    "reportPath": "results",
    "awsRegion": "<region>",
    "auth": {
        "method": "file | environment",
        "credentials": {
            "profile": "<profile-name>"
        }
    }
```

```
}
]
```

All fields that contain values are required as described here:

> **Note**
>
> All paths in this file are defined relative to the *<device-tester-extract-location>*.

`log.location`

The path to the logs folder in the *<device-tester-extract-location>*.

`configFiles.root`

The path to the folder that contains the configuration files.

`configFiles.device`

The path to the `device.json` file.

`testPath`

The path to the folder that contains test suites.

`reportPath`

The path to the folder that will contain test results after IDT runs a test suite.

`awsRegion`

Optional. The AWS region that test suites will use. If not set, then test suites will use the default region specified in each test suite.

`auth.method`

The method IDT uses to retrieve AWS credentials. Supported values are `file` to retrieve credentials from a credentials file, and `environment` to retrieve credentials using environment variables.

`auth.credentials.profile`

The credentials profile to use from the credentials file. This property applies only if `auth.method` is set to `file`.

Debug and run custom test suites

After the required configuration is set, IDT can run your test suite. The runtime of the full test suite depends on the hardware and the composition of the test suite. For reference, it takes approximately 30 minutes to complete the full AWS IoT Greengrass qualification test suite on a Raspberry Pi 3B.

As you write your test suite, you can use IDT to run the test suite in debug mode to check your code before you run it or provide it to test runners.

Run IDT in debug mode

Because test suites depend on IDT to interact with devices, provide the context, and receive results, you cannot simply debug your test suites in an IDE without any IDT interaction. To do so, the IDT CLI provides the `debug-test-suite` command that lets you run IDT in debug mode. Run the following command to view the available options for `debug-test-suite`:

```
devicetester_[linux | mac | win_x86-64] debug-test-suite -h
```

When you run IDT in debug mode, IDT does not actually launch the test suite or run the test orchestrator; instead, it interacts with your IDE to responds to requests made from the test suite running in the IDE and prints the logs to the console. IDT does not time out and waits to exit until manually interrupted. In debug mode, IDT also does not run the test orchestrator and will not generate any report files. To debug your test suite, you must use your IDE to provide some information that IDT usually obtains from the configuration JSON files. Make sure you provide the following information:

- Environment variables and arguments for each test. IDT will not read this information from `test.json` or `suite.json`.
- Arguments to select resource devices. IDT will not read this information from `test.json`.

To debug your test suites, complete the following steps:

1. Create the setting configuration files that are required to run the test suite. For example, if your test suite requires the `device.json`, `resource.json`, and `user data.json`, make sure you configure all of them as needed.
2. Run the following command to place IDT in debug mode and select any devices that are required to run the test.

```
devicetester_[linux | mac | win_x86-64] debug-test-suite [options]
```

After you run this command, IDT waits for requests from the test suite and then responds to them. IDT also generates the environment variables that are required for the case process for the IDT Client SDK.

3. In your IDE, use the `run` or `debug` configuration to do the following:
   a. Set the values of the IDT-generated environment variables.
   b. Set the value of any environment variables or arguments that you specified in your `test.json` and `suite.json` file.
   c. Set breakpoints as needed.
4. Run the test suite in your IDE.

   You can debug and re-run the test suite as many times as needed. IDT does not time out in debug mode.
5. After you complete debugging, interrupt IDT to exit debug mode.

IDT CLI commands to run tests

The following section describes the IDT CLI commands:

IDT v4.0.0

`help`

Lists information about the specified command.

`list-groups`

Lists the groups in a given test suite.

`list-suites`

Lists the available test suites.

`list-supported-products`

Lists the supported products for your version of IDT, in this case AWS IoT Greengrass versions, and AWS IoT Greengrass qualification test suite versions available for the current IDT version.

`list-test-cases`

Lists the test cases in a given test group. The following option is supported:

- `group-id`. The test group to search for. This option is required and must specify a single group.

`run-suite`

Runs a suite of tests on a pool of devices. The following are some commonly used options:

- `suite-id`. The test suite version to run. If not specified, IDT uses the latest version in the `tests` folder.
- `group-id`. The test groups to run, as a comma-separated list. If not specified, IDT runs all test groups in the test suite.
- `test-id`. The test cases to run, as a comma-separated list. When specified, `group-id` must specify a single group.
- `pool-id`. The device pool to test. Test runners must specify a pool if they have multiple device pools defined in your `device.json` file.
- `timeout-multiplier`. Configures IDT to modify the test execution timeout specified in the `test.json` file for a test with a user-defined multiplier.
- `stop-on-first-failure`. Configures IDT to stop execution on the first failure. This option should be used with `group-id` to debug the specified test groups.
- `userdata`. Sets the file that contains user data information required to run the test suite. This is required only if `userdataRequired` is set to true in the `suite.json` file for the test suite.

For more information about `run-suite` options, use the `help` option:

```
devicetester_[linux | mac | win_x86-64] run-suite -h
```

`debug-test-suite`

Run the test suite in debug mode. For more information, see Run IDT in debug mode.

Review IDT test results and logs

This section describes the format in which IDT generates console logs and test reports.

Console message format

AWS IoT Device Tester uses a standard format for printing messages to the console when it starts a test suite. The following excerpt shows an example of a console message generated by IDT.

```
time="2000-01-02T03:04:05-07:00" level=info msg=Using suite: MyTestSuite_1.0.0
executionId=9a52f362-1227-11eb-86c9-8c8590419f30
```

Most console messages consist of the following fields:

`time`

A full ISO 8601 timestamp for the logged event.

`level`

The message level for the logged event. Typically, the logged message level is one of `info`, `warn`, or `error`. IDT issues a `fatal` or `panic` message if it encounters an expected event that causes it to exit early.

`msg`

The logged message.

`executionId`

A unique ID string for the current IDT process. This ID is used to differentiate between individual IDT runs.

Console messages generated from a test suite provide additional information about the device under test and the test suite, test group, and test cases that IDT runs. The following excerpt shows an example of a console message generated from a test suite.

```
time="2000-01-02T03:04:05-07:00" level=info msg=Hello world! suiteId=MyTestSuite
groupId=myTestGroup testCaseId=myTestCase deviceId=my-device
executionId=9a52f362-1227-11eb-86c9-8c8590419f30
```

The test-suite specific part of the console message contains the following fields:

`suiteId`

The name of the test suite currently running.

`groupId`

The ID of the test group currently running.

`testCaseId`

The ID of the test case current running.

`deviceId`

A ID of the device under test that the current test case is using.

To print a test summary to the console when a IDT finishes running a test, you must include a `Report` state in your test orchestrator. The test summary contains information about the test suite, the test results for each group that was run, and the locations of the generated logs and report files. The following example shows a test summary message.

```
========== Test Summary ==========
Execution Time:     5m00s
Tests Completed:    4
Tests Passed:       3
Tests Failed:       1
Tests Skipped:      0
----------------------------------
Test Groups:
    GroupA:         PASSED
    GroupB:         FAILED
----------------------------------
Failed Tests:
    Group Name: GroupB
        Test Name: TestB1
            Reason: Something bad happened
----------------------------------
Path to IoT Device Tester Report: /path/to/awsiotdevicetester_report.xml
Path to Test Execution Logs: /path/to/logs
Path to Aggregated JUnit Report: /path/to/MyTestSuite_Report.xml
```

AWS IoT Device Tester report schema

`awsiotdevicetester_report.xml` is a signed report that contains the following information:

- The IDT version.

- The test suite version.
- The report signature and key used to sign the report.
- The device SKU and the device pool name specified in the `device.json` file.
- The product version and the device features that were tested.
- The aggregate summary of test results. This information is the same as that contained in the *`suite-name`*`_report.xml` file.

```
<apnreport>
    <awsiotdevicetesterversion>idt-version</awsiotdevicetesterversion>
    <testsuiteversion>test-suite-version</testsuiteversion>
    <signature>signature</signature>
    <keyname>keyname</keyname>
    <session>
        <testsession>execution-id</testsession>
        <starttime>start-time</starttime>
        <endtime>end-time</endtime>
    </session>
    <awsproduct>
        <name>product-name</name>
        <version>product-version</version>
        <features>
            <feature name="<feature-name>" value="supported | not-supported | <feature-
value>" type="optional | required"/>
        </features>
    </awsproduct>
    <device>
        <sku>device-sku</sku>
        <name>device-name</name>
        <features>
            <feature name="<feature-name>" value="<feature-value>"/>
        </features>
        <executionMethod>ssh | uart | docker</executionMethod>
    </device>
    <devenvironment>
        <os name="<os-name>"/>
    </devenvironment>
    <report>
        <suite-name-report-contents>
    </report>
</apnreport>
```

The `awsiotdevicetester_report.xml` file contains an `<awsproduct>` tag that contains information about the product being tested and the product features that were validated after running a suite of tests.

Attributes used in the `<awsproduct>` tag

`name`

The name of the product being tested.

`version`

The version of the product being tested.

`features`

The features validated. Features marked as `required` are required for the test suite to validate the device. The following snippet shows how this information appears in the `awsiotdevicetester_report.xml` file.

```
<feature name="ssh" value="supported" type="required"></feature>
```

Features marked as `optional` are not required for validation. The following snippets show optional features.

```
<feature name="hsi" value="supported" type="optional"></feature>

<feature name="mqtt" value="not-supported" type="optional"></feature>
```

Test suite report schema

The *`suite-name`*`_Result.xml` report is in JUnit XML format. You can integrate it into continuous integration and deployment platforms like Jenkins, Bamboo, and so on. The report contains an aggregate summary of test results.

```
<testsuites name="<suite-name> results" time="<run-duration>" tests="<number-of-test>"
 failures="<number-of-tests>" skipped="<number-of-tests>" errors="<number-of-tests>"
 disabled="0">
    <testsuite name="<test-group-id>" package="" tests="<number-of-tests>"
 failures="<number-of-tests>" skipped="<number-of-tests>" errors="<number-of-tests>"
 disabled="0">
```

```
        <!--success-->
        <testcase classname="<classname>" name="<name>" time="<run-duration>"/>
        <!--failure-->
        <testcase classname="<classname>" name="<name>" time="<run-duration>">
            <failure type="<failure-type>">
                reason
            </failure>
        </testcase>
        <!--skipped-->
        <testcase classname="<classname>" name="<name>" time="<run-duration>">
            <skipped>
                reason
            </skipped>
        </testcase>
        <!--error-->
        <testcase classname="<classname>" name="<name>" time="<run-duration>">
            <error>
                reason
            </error>
        </testcase>
    </testsuite>
</testsuites>
```

The report section in both the `awsiotdevicetester_report.xml` or *`suite-name`*`_report.xml` lists the tests that were run and the results.

The first XML tag `<testsuites>` contains the summary of the test execution. For example:

```
<testsuites name="MyTestSuite results" time="2299" tests="28" failures="0" errors="0"
 disabled="0">
```

Attributes used in the `<testsuites>` tag

`name`

The name of the test suite.

`time`

The time, in seconds, it took to run the test suite.

`tests`

The number of tests executed.

`failures`

The number of tests that were run, but did not pass.

`errors`

The number of tests that IDT couldn't execute.

`disabled`

This attribute is not used and can be ignored.

In the event of test failures or errors, you can identify the test that failed by reviewing the `<testsuites>` XML tags. The `<testsuite>` XML tags inside the `<testsuites>` tag show the test result summary for a test group. For example:

```
<testsuite name="combination" package="" tests="1" failures="0" time="161" disabled="0"
 errors="0" skipped="0">
```

The format is similar to the `<testsuites>` tag, but with a `skipped` attribute that is not used and can be ignored. Inside each `<testsuite>` XML tag, there are `<testcase>` tags for each executed test for a test group. For example:

```
<testcase classname="Security Test" name="IP Change Tests" attempts="1"></testcase>>
```

Attributes used in the `<testcase>` tag

`name`

The name of the test.

`attempts`

The number of times IDT executed the test case.

When a test fails or an error occurs, `<failure>` or `<error>` tags are added to the `<testcase>` tag with information for troubleshooting. For example:

```
<testcase classname="mcu.Full_MQTT" name="MQTT_TestCase" attempts="1">
 <failure type="Failure">Reason for the test failure</failure>
 <error>Reason for the test execution error</error>
</testcase>
```

IDT usage metrics

If you provide AWS credentials with required permissions, AWS IoT Device Tester collects and submits usage metrics to AWS. This is an opt-in feature and is used to improve IDT functionality. IDT collects information such as the following:

- The AWS account ID used to run IDT
- The IDT AWS CLI commands used to run tests
- The test suites that are run
- The test suites in the *<device-tester-extract-location>* folder
- The number of devices configured in the device pool
- Test case names and run times
- Test result information, such as whether tests passed, failed, encountered errors, or were skipped
- Product features tested
- IDT exit behavior, such as unexpected or early exits

All of the information that IDT sends is also logged to a `metrics.log` file in the *<device-tester-extract-location>*`/results/`*<execution-id>*`/` folder. You can view the log file to see the information that was collected during a test run. This file is generated only if you choose to collect usage metrics.

To disable metrics collection, you do not need to take additional action. Simply do not store your AWS credentials, and if you do have stored AWS credentials, do not configure the `config.json` file to access them.

Configure your AWS credentials

If you do not already have an AWS account, you must create one. If you already have an AWS account, you simply need to configure the required permissions for your account that allow IDT to send usage metrics to AWS on your behalf.

Step 2: Configure permissions for IDT

In this step, configure the permissions that IDT uses to run tests and collect IDT usage data. You can use the AWS Management Console or AWS Command Line Interface (AWS CLI) to create an IAM policy and a user for IDT, and then attach policies to the user.

- To Configure Permissions for IDT (Console)
- To Configure Permissions for IDT (AWS CLI)

To configure permissions for IDT (console)

Follow these steps to use the console to configure permissions for IDT for AWS IoT Greengrass.

1. Sign in to the IAM console.
2. Create a customer managed policy that grants permissions to create roles with specific permissions.

   a. In the navigation pane, choose **Policies**, and then choose **Create policy**.

   b. On the **JSON** tab, replace the placeholder content with the following policy.

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot-device-tester:SendMetrics"
            ],
            "Resource": "*"
        }
    ]
}
```

   c. Choose **Review policy**.

   d. For **Name**, enter **`IDTUsageMetricsIAMPermissions`**. Under **Summary**, review the permissions granted by your policy.

   e. Choose **Create policy**.

3. Create an IAM user and attach permissions to the user.

   a. Create an IAM user. Follow steps 1 through 5 in Creating IAM users (console) in the *IAM User Guide*. If you already created an IAM user, skip to the next step.

   b. Attach the permissions to your IAM user:

i. On the **Set permissions** page, choose **Attach existing policies to user directly**.

ii. Search for the **IDTUsageMetricsIAMPermissions** policy that you created in the previous step. Select the check box.

c. Choose **Next: Tags**.

d. Choose **Next: Review** to view a summary of your choices.

e. Choose **Create user**.

f. To view the user's access keys (access key IDs and secret access keys), choose **Show** next to the password and access key. To save the access keys, choose **Download.csv** and save the file to a secure location. You use this information later to configure your AWS credentials file.

To configure permissions for IDT (AWS CLI)

Follow these steps to use the AWS CLI to configure permissions for IDT for AWS IoT Greengrass.

1. On your computer, install and configure the AWS CLI if it's not already installed. Follow the steps in Installing the AWS CLI in the *AWS Command Line Interface User Guide*.

   > **Note**
   >
   > The AWS CLI is an open source tool that you can use to interact with AWS services from your command-line shell.

2. Create the following customer managed policy that grants permissions to manage IDT and AWS IoT Greengrass roles.

   Linux or Unix

   ```
   aws iam create-policy --policy-name IDTUsageMetricsIAMPermissions --policy-
   document '{
       "Version": "2012-10-17",
       "Statement": [
           {
               "Effect": "Allow",
               "Action": [
                   "iot-device-tester:SendMetrics"
   ```

```
            ],
            "Resource": "*"
        }
    ]
}'
```

Windows command prompt

```
aws iam create-policy --policy-name IDTUsageMetricsIAMPermissions --policy-
document
                                          '{\"Version\": \"2012-10-17\",
      \"Statement\": [{\"Effect\": \"Allow\", \"Action\": [\"iot-device-
tester:SendMetrics\"], \"Resource\": \"*\"}]}'
```

> **Note**
>
> This step includes a Windows command prompt example because it uses a different JSON syntax than Linux, macOS, or Unix terminal commands.

PowerShell

```
aws iam create-policy --policy-name IDTUsageMetricsIAMPermissions --policy-
document '{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "iot-device-tester:SendMetrics"
            ],
            "Resource": "*"
        }
    ]
}'
```

3. Create an IAM user and attach the permissions required by IDT for AWS IoT Greengrass.

   a. Create an IAM user.

   ```
   aws iam create-user --user-name user-name
   ```

b. Attach the `IDTUsageMetricsIAMPermissions` policy you created to your IAM user. Replace *user-name* with your IAM user name and *<account-id>* in the command with the ID of your AWS account.

```
aws iam attach-user-policy --user-name user-name --policy-arn
 arn:aws:iam::<account-id>:policy/IDTGreengrassIAMPermissions
```

4. Create a secret access key for the user.

```
aws iam create-access-key --user-name user-name
```

Store the output in a secure location. You use this information later to configure your AWS credentials file.

Provide AWS credentials to IDT

To allow IDT to access your AWS credentials and submit metrics to AWS, do the following:

1. Store the AWS credentials for your IAM user as environment variables or in a credentials file:

a. To use environment variables, run the following commands.

Linux or Unix

```
export AWS_ACCESS_KEY_ID=access-key
export AWS_SECRET_ACCESS_KEY=secret-access-key
```

Windows Command Prompt (CMD)

```
set AWS_ACCESS_KEY_ID=access-key
set AWS_SECRET_ACCESS_KEY=secret-access-key
```

PowerShell

```
$env:AWS_ACCESS_KEY_ID="access-key"
$env:AWS_SECRET_ACCESS_KEY="secret-access-key"
```

b. To use the credentials file, add the following information to the `~/.aws/credentials` file.

```
[profile-name]
aws_access_key_id=access-key
aws_secret_access_key=secret-access-key
```

2. Configure the `auth` section of the `config.json` file. For more information, see (Optional) Configure config.json.

Troubleshooting IDT for AWS IoT Greengrass V2

IDT for AWS IoT Greengrass V2 writes errors to various locations based on the type of errors. IDT writes errors to the console, log files, and test reports.

Where to look for errors

High-level errors are displayed on the console while the test is running, and a summary of the failed tests displays when all tests are complete. `awsiotdevicetester_report.xml` contains a summary of all the errors that caused a test to fail. IDT stores the log files for each test run in a directory with a UUID for the test execution, displayed on the console during the test run.

The IDT test logs directory is *<device-tester-extract-location>*`/results/`*<execution-id>*`/logs/`. This directory contains the following files displayed in the table. This is useful for debugging.

File	Description
`test_manager.log`	The logs written to the console while the test was running. The summary of the results at the end of this file includes a list of which tests failed. The warning and error logs in this file can give you some information about the failures.
test-group-id `/`*test-case-id* `/`*test-name* `.log`	Detailed logs for the specific test in a test group. For tests that deploy Greengrass components, the test case log file is called `greengrass-test-run.log`.

File	Description
test-group-id /*test-case-id* / greengrass.log	Detailed logs for AWS IoT Greengrass Core software. IDT copies this file from the device under test when it runs tests that install AWS IoT Greengrass Core software on the device. For more information about the messages in this log file, see Troubleshooting AWS IoT Greengrass V2.
test-group-id /*test-case-id*/*component-name* .log	Detailed logs for Greengrass components that are deployed during test runs. IDT copies component log files from the device under test when it runs tests that deploys specific components. The name of each component log file corresponds to the name of the deployed component. For more informati on about the messages in this log file, see Troubleshooting AWS IoT Greengrass V2.

Resolving IDT for AWS IoT Greengrass V2 errors

Before you run IDT for AWS IoT Greengrass, get the correct configuration files in place. If you receive parsing and configuration errors, your first step is to locate and use a configuration template appropriate for your environment.

If you are still having issues, see the following debugging process.

Topics

- Alias resolution errors
- Conflict errors
- Could not start test error
- Docker qualification image exists errors
- Failed to read credential
- Guice errors with PreInstalled Greengrass
- Invalid signature exception

- Machine learning qualification errors
- Open Test Framework (OTF) failed deployments
- Parsing errors
- Permission denied errors
- Qualification report generation error
- Required parameter missing error
- Security exception on macOS
- SSH connection errors
- Stream manager qualification errors
- Timeout errors
- Version check errors

Alias resolution errors

When you run custom test suites, you might see the following error in the console and in the `test_manager.log`.

```
Couldn't resolve placeholders: couldn't do a json lookup: index out of range
```

This error can occur when the aliases configured in the IDT test orchestrator don't resolve correctly or if the resolved values aren't present in the configuration files. To resolve this error, make sure that your `device.json` and `userdata.json` contain the correct information required for your test suite. For information about the configuration required for AWS IoT Greengrass qualification, see Configure IDT settings to run the AWS IoT Greengrass qualification suite.

Conflict errors

You might see the following error when you run the AWS IoT Greengrass qualification suite concurrently on more than one device.

```
ConflictException: Component [com.example.IDTHelloWorld : 1.0.0] for account [account-
id] already exists with state: [DEPLOYABLE] { RespMetadata: { StatusCode: 409,
 RequestID: “id” }, Message_: “Component [com.example.IDTHelloWorld : 1.0.0] for
 account [account-id] already exists with state: [DEPLOYABLE]” }
```

Concurrent test execution isn't yet supported for the AWS IoT Greengrass qualification suite. Run the qualification suite sequentially for each device.

Could not start test error

You might encounter errors that point to failures that occurred when the test was attempting to start. There are several possible causes, so do the following:

- Make sure that the pool name in your execution command actually exists. IDT references the pool name directly from your `device.json` file.
- Make sure that the devices in your pool have correct configuration parameters.

Docker qualification image exists errors

The Docker application manager qualification tests use the `amazon/amazon-ec2-metadata-mock` container image in Amazon ECR to qualify the device under test.

You might receive the following error if the image is already present in a Docker container on the device under test.

```
The Docker image amazon/amazon-ec2-metadata-mock:version already exists on the device.
```

If you previously downloaded this image and ran the `amazon/amazon-ec2-metadata-mock` container on your device, make sure you remove this image from the device under test before you run the qualification tests.

Failed to read credential

When testing Windows devices, you might encounter the `Failed to read credential` error in the `greengrass.log` file if the user that you use to connect to the device under test is not set up in the credential manager on that device.

To resolve this error, configure the user and password for the IDT user in the credential manager on the device under test.

For more information, see Configure user credentials for Windows devices.

Guice errors with PreInstalled Greengrass

While running IDT with PreInstalled Greengrass, if you encounter an error of `Guice` or `ErrorInCustomProvider`, check if the file `userdata.json` has the `InstalledDirRootOnDevice` set to the Greengrass installation folder. IDT checks for the file `effectiveConfig.yaml` under `<InstallationDirRootOnDevice>/config/effectiveConfig.yaml`.

For more information, see Configure user credentials for Windows devices.

Invalid signature exception

When you run Lambda qualification tests, you might encounter the `invalidsignatureexception` error if your IDT host machine experiences network access issues. Reset your router and run the tests again.

Machine learning qualification errors

When you run machine learning (ML) qualification tests, you might encounter qualification failures if your device doesn't meet the requirements to deploy the AWS-provided ML components. To troubleshoot ML qualification errors, do the following:

- Look for error details in the component logs for the components that were deployed during the test run. Component logs are located in the *<device-tester-extract-location>*/results/*<execution-id>*/logs/*<test-group-id>* directory.
- Add the `-Dgg.persist=installed.software` argument to the `test.json` file for the failing test case. The `test.json` file is located in the *<device-tester-extract-location>*/tests/GGV2Q_*version* directory.

Open Test Framework (OTF) failed deployments

If OTF tests fail to complete the deployment, a likely cause may be the permissions set for the parent folder of `TempResourcesDirOnDevice` and `InstallationDirRootOnDevice`. To set this folder's permissions correctly, run the following command. Replace *folder-name* with the name of the parent folder.

```
sudo chmod 755 folder-name
```

Parsing errors

Typos in a JSON configuration can lead to parsing errors. Most of the time, the issue is a result of omitting a bracket, comma, or quotation mark from your JSON file. IDT performs JSON validation and prints debugging information. It prints the line where the error occurred, the line number, and the column number of the syntax error. This information should be enough to help you fix the error, but if you still can't locate the error, you can perform validation manually in your IDE, a text editor such as Atom or Sublime, or through an online tool like JSONLint.

Permission denied errors

IDT performs operations on various directories and files in a device under test. Some of these operations require root access. To automate these operations, IDT must be able to run commands with sudo without typing a password.

Follow these steps to allow sudo access without typing a password.

> **Note**
>
> `user` and `username` refer to the SSH user used by IDT to access the device under test.

1. Use **sudo usermod -aG sudo *<ssh-username>*** to add your SSH user to the sudo group.
2. Sign out and then sign in for changes to take effect.
3. Open `/etc/sudoers` file and add the following line to the end of the file: *<ssh-username>* `ALL=(ALL) NOPASSWD: ALL`

   > **Note**
   >
   > As a best practice, we recommend that you use **sudo visudo** when you edit `/etc/sudoers`.

Qualification report generation error

IDT supports the four latest *major*.*minor* versions of the AWS IoT Greengrass V2 qualification suite (GGV2Q) to generate qualification reports that you can submit to AWS Partner Network to include your devices in the AWS Partner Device Catalog. Earlier versions of the qualification suite don’t generate qualification reports.

If you have questions about the support policy, contact AWS Support.

Required parameter missing error

When IDT adds new features, it might introduce changes to the configuration files. Using an old configuration file might break your configuration. If this happens, the *<test_case_id>*.log file under /results/*<execution-id>*/logs explicitly lists all missing parameters. IDT also validates your JSON configuration file schemas to verify that you are using the latest supported version.

Security exception on macOS

When you run IDT on a macOS host computer, it blocks IDT from running. To run IDT, grant a security exception to the executables that is part of IDT runtime functionality. When you see the warning message display on your host computer, do the following for each of the applicable executables:

To grant a security exception to IDT executables

1. On the macOS computer, on the Apple menu, open **System Preferences**.
2. Choose **Security & Privacy**, then on the **General** tab, choose the lock icon to make changes to security settings.
3. In case of blocked devicetester_mac_x86-64, look for the message "devicetester_mac_x86-64" was blocked from use because it is not from an identified developer. and choose **Allow Anyway**.
4. Resume IDT testing, until you get through all executables involved.

SSH connection errors

When IDT can't connect to a device under test, it logs connection failures in /results/*<execution-id>*/logs/*<test-case-id>*.log. SSH messages appear at the top of this log file because connecting to a device under test is one of the first operations that IDT performs.

Most Windows configurations use the PuTTy terminal application to connect to Linux hosts. This application requires that you convert standard PEM private key files into a proprietary Windows format called PPK. If you configure SSH in your device.json file, use PEM files. If you use a PPK file, IDT can't create an SSH connection with the AWS IoT Greengrass device and can't run tests.

Starting with IDT v4.4.0, if you haven't enabled SFTP on your device under test, then you might see the following error in the log file.

```
SSH connection failed with EOF
```

To resolve this error, enable SFTP on your device.

Stream manager qualification errors

When you run stream manager qualification tests, you might see the following error in the `com.aws.StreamManagerExport.log` file.

```
Failed to upload data to S3
```

This error can occur when stream manager uses the AWS credentials in the `~/root/.aws/credentials` file on your device instead of using the environment credentials that IDT exports to the device under test. To prevent this issue, delete the `credentials` file on your device, and re-run the qualification test.

Timeout errors

You can increase the timeout for each test by specifying a timeout multiplier applied to the default value of each test's timeout. Any value configured for this flag must be greater than or equal to 1.0.

To use the timeout multiplier, use the flag `--timeout-multiplier` when running the tests. For example:

```
./devicetester_linux run-suite --suite-id GGV2Q_1.0.0 --pool-id DevicePool1 --timeout-
multiplier 2.5
```

For more information, run `run-suite --help`.

Some timeout errors occur when IDT test cases can’t be completed because of configuration issues. You can’t resolve these errors by increasing the timeout multiplier. Use the logs from the test run to troubleshoot the underlying configuration issues.

- If the MQTT or Lambda component logs contain `Access denied` errors, your Greengrass installation folder might not have the correct file permissions. Run the following command for each folder in the installation path that you defined in your `userdata.json` file.

```
sudo chmod 755 folder-name
```

- If the Greengrass logs indicate that the Greengrass CLI deployment isn't complete, do the following:
  - Verify that `bash` is installed on the device under test.
  - If your `userdata.json` file includes the `GreengrassCliVersion` configuration parameter, remove it. This parameter is deprecated in IDT v4.1.0 and later versions. For more information, see Configure userdata.json.
- If the Lambda deployment test failed with an error message of "Validating Lambda publish: time out" and you receive an error in the test log file (`idt-gg2-lambda-function-idt-<resource-id>.log`) that says `Error: Could not find or load main class com.amazonaws.greengrass.runtime.LambdaRuntime.`, do the following:
  - Verify what folder was used for `InstallationDirRootOnDevice` in the `userdata.json` file.
  - Make sure the correct user permissions are set up on your device. For more details, see Configure user permissions on your device.

Version check errors

IDT issues the following error when the AWS user credentials for the IDT user don't have the required IAM permissions.

```
Failed to check version compatibility
```

The AWS user that doesn't have the required IAM permissions.

Support policy for AWS IoT Device Tester for AWS IoT Greengrass

AWS IoT Device Tester for AWS IoT Greengrass is a test automation tool used to validate and qualify your AWS IoT Greengrass devices for inclusion in the AWS Partner Device Catalog. We recommend that you use the most recent version of AWS IoT Greengrass and AWS IoT Device Tester to test or qualify your devices.

At least one version of AWS IoT Device Tester is available for each supported version of AWS IoT Greengrass. For supported versions of AWS IoT Greengrass, see Greengrass nucleus versions. For supported versions of AWS IoT Device Tester, see Supported versions of AWS IoT Device Tester for AWS IoT Greengrass V2.

You can also use any of the supported versions of AWS IoT Greengrass and AWS IoT Device Tester to test or qualify your devices. Although you can continue to use unsupported versions of AWS IoT Device Tester, those versions do not receive bug fixes or updates. If you have questions about the support policy, contact AWS Support.

Greengrass based IoT solutions

Eurotech's Everyware GreenEdge is in a preview release for AWS IoT Greengrass and is subject to change. This solution is not supported by AWS. You must contact Eurotech for any issues with this device.

AWS IoT Greengrass offers solutions from Partners to optimize your experience installing Greengrass. The following is a solution that AWS has partnered with Eurotech to offer. This solution comes with AWS IoT Greengrass Core edge runtime and additional capabilities pre-installed.

Eurotech

AWS has partnered with Eurotech to offer an IoT solution for customers who are looking for a device that comes with AWS IoT Greengrass Core software pre-installed. Eurotech's Everyware GreenEdge is an IoT edge software that is pre-configured and pre-qualified by AWS. This solution merges the capabilities of Greengrass and the Eurotech Everyware Software Framework (ESF) to offer customers extensive southbound connectivity through protocol adapters like: Modbus, OPC-UA Client/Server, S7, TwinCAT, J1939, DNP3 Master/Outstation, and more. With this solution, you can also send data to the AWS Cloud and connect to all northbound AWS services (such as AWS IoT Core, AWS IoT SiteWise, Amazon S3, and Amazon Kinesis Video Streams). Combined with Everyware Cloud, Eurotech's device management solution, this solution introduces a novel Zero-Touch Provisioning service, which simplifies device onboarding and mass deployment.

For more information about Eurotech, see Eurotech.

Troubleshooting AWS IoT Greengrass V2

Use the troubleshooting information and solutions in this section to help resolve issues with AWS IoT Greengrass Version 2.

Topics

- View AWS IoT Greengrass Core software and component logs
- AWS IoT Greengrass Core software issues
- AWS IoT Greengrass cloud issues
- Core device deployment issues
- Core device component issues
- Core device Lambda function component issues
- Component version discontinued
- Greengrass Command Line Interface issues
- AWS Command Line Interface issues
- Detailed deployment error codes
- Detailed component status codes

View AWS IoT Greengrass Core software and component logs

The AWS IoT Greengrass Core software writes logs to the local file system that you can use to view real-time information about the core device. You can also configure core devices to write logs to CloudWatch Logs, so you can remotely troubleshoot core devices. These logs can help you identify issues with components, deployments, and core devices. For more information, see Monitor AWS IoT Greengrass logs.

AWS IoT Greengrass Core software issues

Troubleshoot AWS IoT Greengrass Core software issues.

Topics

- ThrottlingException from ListDeployments API

- Unable to set up core device
- Unable to start the AWS IoT Greengrass Core software as a system service
- Unable to set up nucleus as a system service
- Unable to connect to AWS IoT Core
- Out of memory error
- Unable to install Greengrass CLI
- User root is not allowed to execute
- com.aws.greengrass.lifecyclemanager.GenericExternalService: Could not determine user/group to run with
- Failed to map segment from shared object: operation not permitted
- Failed to set up Windows service
- com.aws.greengrass.util.exceptions.TLSAuthException: Failed to get trust manager
- com.aws.greengrass.deployment.IotJobsHelper: No connection available during subscribing to Iot Jobs descriptions topic. Will retry in sometime
- software.amazon.awssdk.services.iam.model.IamException: The security token included in the request is invalid
- software.amazon.awssdk.services.iot.model.IotException: User: <user> is not authorized to perform: iot:GetPolicy
- Error: com.aws.greengrass.shadowmanager.sync.model.FullShadowSyncRequest: Could not execute cloud shadow get request
- Operation aws.greengrass#<operation> is not supported by Greengrass
- java.io.FileNotFoundException: <stream-manager-store-root-dir>/stream_manager_metadata_store (Permission denied)
- com.aws.greengrass.security.provider.pkcs11.PKCS11CryptoKeyService: Private key or certificate with label <label> does not exist
- software.amazon.awssdk.services.secretsmanager.model.SecretsManagerException: User: <user> is not authorized to perform: secretsmanager:GetSecretValue on resource: <arn>
- software.amazon.awssdk.services.secretsmanager.model.SecretsManagerException: Access to KMS is not allowed
- java.lang.NoClassDefFoundError: com/aws/greengrass/security/CryptoKeySpi
- com.aws.greengrass.security.provider.pkcs11.PKCS11CryptoKeyService: CKR_OPERATION_NOT_INITIALIZED

- Greengrass core device stuck on nucleus v2.12.3
- Greengrass nucleus v2.14.0 systemd template issue

ThrottlingException from ListDeployments API

`ThrottlingException` from `ListDeployments` API: You might see this when you have a large number of deployments in account.

To solve this, do one of following:

- If you use SDK, please specify MaxResult parameter. For example, for JavaSDK with a small value (e.g. 5).
- You can use AWS Service Quotas to request a limit increase of the rate limit of the`DescribeJob` API. You can go to the Service quota console, select quotas of AWS IoT and the limit name is **DescribeJob throttle limit**. You can increase it from 10 to 50.

Unable to set up core device

If the AWS IoT Greengrass Core software installer fails and you aren't able to set up a core device, you might need to uninstall the software and try again. For more information, see Uninstall the AWS IoT Greengrass Core software.

Unable to start the AWS IoT Greengrass Core software as a system service

If the AWS IoT Greengrass Core software fails to start, check the system service logs to identify the issue. One common issue is where Java isn't available on the PATH environment variable (Linux) or PATH system variable (Windows).

Unable to set up nucleus as a system service

You might see this error when the AWS IoT Greengrass Core software installer fails to set up AWS IoT Greengrass as a system service. On Linux devices, this error typically occurs if the core device doesn't have the systemd init system. The installer can successfully set up the AWS IoT Greengrass Core software even if it fails to set up the system service.

Do one of the following:

- Configure and run the AWS IoT Greengrass Core software as a system service. You must configure the software as a system service to use all of the features of AWS IoT Greengrass. You can install systemd or use a different init system. For more information, see Configure the Greengrass nucleus as a system service.
- Run the AWS IoT Greengrass Core software without a system service. You can run the software using a loader script that the installer sets up in the Greengrass root folder. For more information, see Run the AWS IoT Greengrass Core software without a system service.

Unable to connect to AWS IoT Core

You might see this error when the AWS IoT Greengrass Core software can't connect to AWS IoT Core to retrieve deployment jobs, for example. Do the following:

- Check that your core device can connect to the internet and AWS IoT Core. For more information about the AWS IoT Core endpoint to which your device connects, see Configure the AWS IoT Greengrass Core software.
- Check that your core device's AWS IoT thing uses a certificate that allows the `iot:Connect`, `iot:Publish`, `iot:Receive`, and `iot:Subscribe` permissions.
- If your core device uses a network proxy, check that your core device has a device role and that its role allows the `iot:Connect`, `iot:Publish`, `iot:Receive`, and `iot:Subscribe` permissions.

Out of memory error

This error typically occurs if your device doesn't have sufficient memory to allocate an object in the Java heap. On devices with limited memory, you might need to specify a maximum heap size to control memory allocation. For more information, see Control memory allocation with JVM options.

Unable to install Greengrass CLI

You might see the following console message when you use the `--deploy-dev-tools` argument in your installation command for AWS IoT Greengrass Core.

```
Thing group exists, it could have existing deployment and devices, hence NOT creating
 deployment for Greengrass first party dev tools, please manually create a deployment
 if you wish to
```

This occurs when the Greengrass CLI component is not installed because your core device is a member of a thing group that has an existing deployment. If you see this message, you can manually deploy the Greengrass CLI component (`aws.greengrass.Cli`) to the device to install the Greengrass CLI. For more information, see Install the Greengrass CLI.

User root is not allowed to execute

You might see this error when the user that runs the AWS IoT Greengrass Core software (typically `root`) doesn't have permission to run `sudo` with any user and any group. For the default `ggc_user` system user, this error looks like the following:

```
Sorry, user root is not allowed to execute <command> as ggc_user:ggc_group.
```

Check that your `/etc/sudoers` file gives the user permission to run `sudo` as other groups. The permission for the user in `/etc/sudoers` should look like the following example.

```
root    ALL=(ALL:ALL) ALL
```

com.aws.greengrass.lifecyclemanager.GenericExternalService: Could not determine user/group to run with

You might see this error when the core device tries to run a component, and the Greengrass nucleus doesn't specify a default system user to use to run components.

To fix this issue, configure the Greengrass nucleus to specify the default system user that runs components. For more information, see Configure the user that runs components and Configure the default component user.

Failed to map segment from shared object: operation not permitted

You might see this error when the AWS IoT Greengrass Core software fails to start because the `/tmp` folder is mounted with `noexec` permissions. The AWS Common Runtime (CRT) library uses the `/tmp` folder by default.

Do one of the following:

- Run the following command to remount the `/tmp` folder with `exec` permissions and try again.

```
sudo mount -o remount,exec /tmp
```

- If you run Greengrass nucleus v2.5.0 or later, you can set a JVM option to change the folder that the AWS CRT library uses. You can specify the `jvmOptions` parameter in the Greengrass nucleus component configuration in a deployment or when you install the AWS IoT Greengrass Core software. Replace *`/path/to/use`* with the path to a folder that the AWS CRT library can use.

```
{
  "jvmOptions": "-Daws.crt.lib.dir=\"/path/to/use\""
}
```

Failed to set up Windows service

You might see this error if you install the AWS IoT Greengrass Core software on a Microsoft Windows 2016 device. The AWS IoT Greengrass Core software is not supported on Windows 2016, for a list of supported operating systems, see Supported platforms.

If you must use Windows 2016, you can do the following:

1. Unzip the downloaded AWS IoT Greengrass Core installation archive
2. In the `Greengrass` directory open the `bin/greengrass.xml.template` file.
3. Add the `<autoRefresh>` tag to the end of the file just before the `</service>` tag.

```
  </log>
  <autoRefresh>false</autoRefresh>
</service>
```

com.aws.greengrass.util.exceptions.TLSAuthException: Failed to get trust manager

You might see this error when you install the AWS IoT Greengrass Core software without a root certificate authority (CA) file.

```
2022-06-05T10:00:39.556Z [INFO] (main) com.aws.greengrass.lifecyclemanager.Kernel:
 service-loaded. {serviceName=DeploymentService}
```

```
2022-06-05T10:00:39.943Z [WARN] (main)
 com.aws.greengrass.componentmanager.ClientConfigurationUtils: configure-greengrass-
mutual-auth. Error during configure greengrass client mutual auth. {}
com.aws.greengrass.util.exceptions.TLSAuthException: Failed to get trust manager
```

Check that you specify a valid root CA file with the `rootCaPath` parameter in the configuration file that you provide to the installer. For more information, see Install the AWS IoT Greengrass Core software.

com.aws.greengrass.deployment.IotJobsHelper: No connection available during subscribing to Iot Jobs descriptions topic. Will retry in sometime

You might see this warning message when the core device can't connect to AWS IoT Core to subscribe to deployment job notifications. Do the following:

- Check that the core device is connected to the internet and can reach the AWS IoT data endpoint that you configured. For more information about endpoints that core devices use, see Allow device traffic through a proxy or firewall.
- Check the Greengrass logs for other errors that reveal other root causes.

software.amazon.awssdk.services.iam.model.IamException: The security token included in the request is invalid

You might see this error when you install the AWS IoT Greengrass Core software with automatic provisioning, and the installer uses an AWS session token that isn't valid. Do the following:

- If you use temporary security credentials, check that the session token is correct and that you are copying and pasting the complete session token.
- If you use long-term security credentials, check that the device doesn't have a session token from a time where you previously used temporary credentials. Do the following:

  1. Run the following command to unset the session token environment variable.

     Linux or Unix

     ```
     unset AWS_SESSION_TOKEN
     ```

Windows Command Prompt (CMD)

```
set AWS_SESSION_TOKEN=
```

PowerShell

```
Remove-Item Env:\AWS_SESSION_TOKEN
```

2. Check if the AWS credentials file, `~/.aws/credentials`, contains a session token, `aws_session_token`. If so, remove that line from the file.

```
aws_session_token = AQoEXAMPLEH4aoAH0gNCAPyJxz4BlCFFxWNE1OPTgk5TthT
+FvwqnKwRc0IfrRh3c/LTo6UDdyJwOOvEVPvLXCrrrUtdnniCEXAMPLE/
IvU1dYUg2RVAJBanLiHb4IgRmpRV3zrkuWJOgQs8IZZaIv2BXIa2R40lgk
```

You can also install the AWS IoT Greengrass Core software without providing AWS credentials. For more information, see Install AWS IoT Greengrass Core software with manual resource provisioning or Install AWS IoT Greengrass Core software with AWS IoT fleet provisioning.

software.amazon.awssdk.services.iot.model.IotException: User: <user> is not authorized to perform: iot:GetPolicy

You might see this error when you install the AWS IoT Greengrass Core software with automatic provisioning, and the installer uses AWS credentials that don't have the required permissions. For more information about the permissions that are required, see Minimal IAM policy for installer to provision resources.

Check the permissions for the credentials' IAM identity, and grant the IAM identity any required permissions that are missing.

Error: com.aws.greengrass.shadowmanager.sync.model.FullShadowSyncRequest: Could not execute cloud shadow get request

You might see this error when you use the shadow manager component to sync device shadows with AWS IoT Core. The HTTP 403 status code indicates that this error occurred because the core device's AWS IoT policy doesn't grant permission to call `GetThingShadow`.

```
com.aws.greengrass.shadowmanager.sync.model.FullShadowSyncRequest: Could not execute
 cloud shadow get request. {thing name=MyGreengrassCore, shadow name=MyShadow}
2021-07-14T21:09:02.456Z [ERROR] (pool-2-thread-109)
 com.aws.greengrass.shadowmanager.sync.SyncHandler: sync. Skipping sync request. {thing
 name=MyGreengrassCore, shadow name=MyShadow}
com.aws.greengrass.shadowmanager.exception.SkipSyncRequestException:
 software.amazon.awssdk.services.iotdataplane.model.IotDataPlaneException:
 null (Service: IotDataPlane, Status Code: 403, Request ID:
 f6e713ba-1b01-414c-7b78-5beb3f3ad8f6, Extended Request ID: null)
```

To sync local shadows with AWS IoT Core, the core device's AWS IoT policy must grant the following permissions:

- `iot:GetThingShadow`
- `iot:UpdateThingShadow`
- `iot:DeleteThingShadow`

Check the core device's AWS IoT policy, and add any required permissions that are missing. For more information, see the following:

- AWS IoT Core policy actions in the *AWS IoT Developer Guide*
- Update a core device's AWS IoT policy

Operation aws.greengrass#<operation> is not supported by Greengrass

You might see this error when you use an interprocess communication (IPC) operation in a custom Greengrass component, and the required AWS-provided component isn't installed on the core device.

To fix this issue, add the required component as a dependency in your component recipe, so the AWS IoT Greengrass Core software installed the required component when you deploy your component.

- Retrieve secret values – `aws.greengrass.SecretManager`
- Interact with local shadows – `aws.greengrass.ShadowManager`
- Manage local deployments and components – `aws.greengrass.Cli` v2.6.0 or later

- Authenticate and authorize client devices – `aws.greengrass.clientdevices.Auth` v2.2.0 or later

java.io.FileNotFoundException: <stream-manager-store-root-dir>/stream_manager_metadata_store (Permission denied)

You might see this error in the stream manager log file (`aws.greengrass.StreamManager.log`) when you configure stream manager to use a root folder that doesn't exist or have the correct permissions. For more information about how to configure this folder, see stream manager configuration.

com.aws.greengrass.security.provider.pkcs11.PKCS11CryptoKeyService: Private key or certificate with label <label> does not exist

This error occurs when the PKCS#11 provider component can't find or load the private key or certificate that you specify when you configure the AWS IoT Greengrass Core software to use a hardware security module (HSM). Do the following:

- Check that the private key and certificate are stored in the HSM using the slot, user PIN, and object label that you configure the AWS IoT Greengrass Core software to use.
- Check that the private key and certificate use the same object label in the HSM.
- If your HSM supports object IDs, check that the private key and certificate use the same object ID in the HSM.

Check the documentation for your HSM to learn how to query details about the security tokens in the HSM. If you need to change the slot, object label, or object ID for a security token, check the documentation for your HSM to learn how to do so.

software.amazon.awssdk.services.secretsmanager.model.SecretsManagerExc User: <user> is not authorized to perform: secretsmanager:GetSecretValue on resource: <arn>

This error can occur when you use the secret manager component to deploy an AWS Secrets Manager secret. If the core device's token exchange IAM role doesn't grant permission to get the secret, the deployment fails and the Greengrass logs include this error.

To authorize a core device to download a secret

1. Add the `secretsmanager:GetSecretValue` permission to the core device's token exchange role. The following example policy statement grants permission to get the value of a secret.

   ```
   {
       "Effect": "Allow",
       "Action": [
           "secretsmanager:GetSecretValue"
       ],
       "Resource": [
           "arn:aws:secretsmanager:us-west-2:123456789012:secret:MyGreengrassSecret-
   abcdef"
       ]
   }
   ```

   For more information, see Authorize core devices to interact with AWS services.

2. Reapply the deployment to the core device. Do one of the following:

   - Revise the deployment without any changes. The core device tries to download the secret again when it receives the revised deployment. For more information, see Revise deployments.
   - Restart the AWS IoT Greengrass Core software to retry the deployment. For more information, see Run the AWS IoT Greengrass Core software

   The deployment succeeds if secret manager downloads the secret successfully.

software.amazon.awssdk.services.secretsmanager.model.SecretsManagerExc Access to KMS is not allowed

This error can occur when you use the secret manager component to deploy an AWS Secrets Manager secret that is encrypted by an AWS Key Management Service key. If the core device's token exchange IAM role doesn't grant permission to decrypt the secret, the deployment fails and the Greengrass logs include this error.

To fix the issue, add the `kms:Decrypt` permission to the core device's token exchange role. For more information, see the following:

- Secret encryption and decryption in the *AWS Secrets Manager User Guide*

- Authorize core devices to interact with AWS services

java.lang.NoClassDefFoundError: com/aws/greengrass/security/CryptoKeySpi

You might see this error when you try to install the AWS IoT Greengrass Core software with hardware security and you use an earlier Greengrass nucleus version that doesn't support hardware security integration. To use hardware security integration, you must use Greengrass nucleus v2.5.3 or later.

com.aws.greengrass.security.provider.pkcs11.PKCS11CryptoKeyService: CKR_OPERATION_NOT_INITIALIZED

You might see this error when you use the TPM2 library when running AWS IoT Greengrass Core as a system service.

This error indicates that you need to add an environment variable that provides the location of the PKCS#11 store in the AWS IoT Greengrass Core systemd service file.

For more information, see the Requirements section of the PKCS#11 provider component documentation.

Greengrass core device stuck on nucleus v2.12.3

If your Greengrass core device won't revise your deployment from nucleus version 2.12.3, you might need to download and replace the `Greengrass.jar` file with Greengrass nucleus version 2.12.2. Do the following:

1. On your Greengrass core device, run the following command to stop the Greengrass Core software.

   Linux or Unix

   ```
   sudo systemctl stop greengrass
   ```

   Windows Command Prompt (CMD)

   ```
   sc stop "greengrass"
   ```

PowerShell

```
Stop-Service -Name "greengrass"
```

2. On your core device, download the AWS IoT Greengrass software to a file named `greengrass-2.12.2.zip`.

   Linux or Unix

   ```
   curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-2.12.2.zip >
    greengrass-2.12.2.zip
   ```

   Windows Command Prompt (CMD)

   ```
   curl -s https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-2.12.2.zip >
    greengrass-2.12.2.zip
   ```

   PowerShell

   ```
   iwr -Uri https://d2s8p88vqu9w66.cloudfront.net/releases/greengrass-2.12.2.zip -
   OutFile greengrass-2.12.2.zip
   ```

3. Unzip the AWS IoT Greengrass Core software to a folder on your device. Replace *GreengrassInstaller* with the folder that you want to use.

   Linux or Unix

   ```
   unzip greengrass-2.12.2.zip -d GreengrassInstaller && rm greengrass-2.12.2.zip
   ```

   Windows Command Prompt (CMD)

   ```
   mkdir GreengrassInstaller && tar -xf greengrass-2.12.2.zip -
   C GreengrassInstaller && del greengrass-2.12.2.zip
   ```

   PowerShell

   ```
   Expand-Archive -Path greengrass-2.12.2.zip -DestinationPath .\
   \GreengrassInstaller
   rm greengrass-2.12.2.zip
   ```

4. Run the following command to override the nucleus version 2.12.3 Greengrass JAR file with the nucleus version 2.12.2 Greengrass JAR file.

   Linux or Unix

   ```
   sudo cp ./GreengrassInstaller/lib/Greengrass.jar /greengrass/v2/packages/
   artifacts-unarchived/aws.greengrass.Nucleus/2.12.3/aws.greengrass.nucleus/lib
   ```

   Windows Command Prompt (CMD)

   ```
   robocopy ./GreengrassInstaller/lib/Greengrass.jar /greengrass/v2/packages/
   artifacts-unarchived/aws.greengrass.Nucleus/2.12.3/aws.greengrass.nucleus/lib /E
   ```

   PowerShell

   ```
   cp -Path ./GreengrassInstaller/lib/Greengrass.jar -Destination /
   greengrass/v2/packages/artifacts-unarchived/aws.greengrass.Nucleus/2.12.3/
   aws.greengrass.nucleus/lib
   ```

5. Run the following command to start the Greengrass Core software.

   Linux or Unix

   ```
   sudo systemctl start greengrass
   ```

   Windows Command Prompt (CMD)

   ```
   sc start "greengrass"
   ```

   PowerShell

   ```
   Start-Service -Name "greengrass"
   ```

Greengrass nucleus v2.14.0 systemd template issue

You might encounter this issue if you installed Greengrass nucleus v2.14.0 on a Linux device with its default systemd service template. Do the following:

1. On your Greengrass core device, run the following command to revert the systemd service file to the way it is in nucleus v2.13.0 and earlier.

   Linux or Unix

   ```
   sudo sed -i 's|ExecStart=/bin/sh -c "\(.*\) >> .*/logs/loader.log 2>&1"|
   ExecStart=/bin/sh \1|' /etc/systemd/system/greengrass.service
   ```

2. Apply the changes.

   Linux or Unix

   ```
   sudo systemctl daemon-reload
   sudo systemctl restart greengrass
   ```

AWS IoT Greengrass cloud issues

Use the following information to troubleshoot issues with the AWS IoT Greengrass console and API. Each entry corresponds to an error message that you might see when you perform an action.

An error occurred (AccessDeniedException) when calling the CreateComponentVersion operation: User: arn:aws:iam::123456789012:user/<username> is not authorized to perform: null

You might see this error when you create a component version from the AWS IoT Greengrass console or with the CreateComponentVersion operation.

This error indicates that your recipe isn't valid JSON or YAML. Check the syntax of your recipe, fix any syntax issues, and try again. You can use an online JSON or YAML syntax checker to identify syntax issues in your recipe.

Invalid Input: Encountered following errors in Artifacts: {<s3ArtifactUri> = Specified artifact resource cannot be accessed}

You might see this error when you create a component version from the AWS IoT Greengrass console or with the CreateComponentVersion operation. This error indicates that an S3 artifact in the component recipe isn't valid.

Do the following:

- Check that the S3 bucket is in the same AWS Region where you create the component. AWS IoT Greengrass doesn't support cross-Region requests for component artifacts.
- Check that the artifact URI is a valid S3 object URL, and check the artifact exists at that S3 object URL.
- Check that your AWS account has permission to access the artifact at its S3 object URL.

INACTIVE deployment status

You might get an `INACTIVE` deployment status when you call the ListDeployments API without the required dependent AWS IoT policies. You must have the necessary permissions in order to get an accurate deployment status. You can find the dependent actions by looking in the Actions defined by AWS IoT Greengrass V2 and following the permissions needed for `ListDeployments`. Without the required dependent AWS IoT permissions, you will still see the deployment status but you might see an inaccurate deployment status of `INACTIVE`.

CreateDeployment returns a success response but GetDeployment returns a 404 error

You might see this issue when you call the CreateDeployment operation to deploy to an AWS IoT thing or thing group. The `CreateDeployment` call returns a `200` status code with a valid deployment ID. However, when you call GetDeployment with that deployment ID, you receive a `ResourceNotFoundException` error. The AWS IoT Greengrass console shows a "Deployment not found" page for the deployment URL. The previous deployment for the target might also show an `INACTIVE` status with no replacement.

This issue can occur when multiple `CreateDeployment` requests target the same AWS IoT thing at the same time. A race condition can cause the cloud to fail to record the deployment. The core device still receives and applies the deployment through the AWS IoT device shadow, but the AWS IoT Greengrass cloud service has no record of it. After this issue occurs, it might also affect subsequent deployments to the same target.

Do the following to resolve and prevent this issue:

- Avoid sending concurrent `CreateDeployment` requests that target the same AWS IoT thing. Serialize your deployment requests or add a delay between them to ensure that only one `CreateDeployment` call targets a specific thing at a time.

- If this issue already affects your target, subsequent deployments might continue to return 404 errors. To resolve this, delete the stale deployment record for the affected target so that the next deployment can proceed normally.

Core device deployment issues

Troubleshoot deployment issues on Greengrass core devices. Each entry corresponds to a log message that you might see on your core device.

Topics

- Error: com.aws.greengrass.componentmanager.exceptions.PackageDownloadException: Failed to download artifact
- Error: com.aws.greengrass.componentmanager.exceptions.ArtifactChecksumMismatchException: Integrity check for downloaded artifact failed. Probably due to file corruption.
- Error: com.aws.greengrass.componentmanager.exceptions.NoAvailableComponentVersionException: Failed to negotiate component <name> version with cloud and no local applicable version satisfying requirement <requirements>
- software.amazon.awssdk.services.greengrassv2data.model.ResourceNotFoundException: The latest version of Component <componentName> doesn't claim platform <coreDevicePlatform> compatibility
- com.aws.greengrass.componentmanager.exceptions.PackagingException: The deployment attempts to update the nucleus from aws.greengrass.Nucleus-<version> to aws.greengrass.Nucleus-<version> but no component of type nucleus was included as target component
- Error: com.aws.greengrass.deployment.exceptions.DeploymentException: Unable to process deployment. Greengrass launch directory is not set up or Greengrass is not set up as a system service
- Info: com.aws.greengrass.deployment.exceptions.RetryableDeploymentDocumentDownloadException: Greengrass Cloud Service returned an error when getting full deployment configuration
- Warn: com.aws.greengrass.deployment.DeploymentService: Failed to get thing group hierarchy
- Deployment unexpectedly removes components from a core device

- Info: com.aws.greengrass.deployment.DeploymentDocumentDownloader: Calling Greengrass cloud to get full deployment configuration
- Caused by: software.amazon.awssdk.services.greengrassv2data.model.GreengrassV2DataException: null (Service: GreengrassV2Data, Status Code: 403, Request ID: <some_request_id>, Extended Request ID: null)

Error: com.aws.greengrass.componentmanager.exceptions.PackageDownloadExcep Failed to download artifact

You might see this error when the AWS IoT Greengrass Core software fails to download a component artifact when the core device applies a deployment. The deployment fails as a result of this error.

When you receive this error, the log also includes a stack trace that you can use to identify the specific issue. Each of the following entries corresponds to a message that you might see in the stack trace of the `Failed to download artifact` error message.

Topics

- software.amazon.awssdk.services.s3.model.S3Exception: null (Service: S3, Status Code: 403, Request ID: null, ...)
- software.amazon.awssdk.services.s3.model.S3Exception: Access Denied (Service: S3, Status Code: 403, Request ID: <requestID>

software.amazon.awssdk.services.s3.model.S3Exception: null (Service: S3, Status Code: 403, Request ID: null, ...)

The PackageDownloadException error might include this stack trace in the following cases:

- The component artifact isn't available at the S3 object URL that you specify in the component's recipe. Check that you uploaded the artifact to the S3 bucket and that the artifact URI matches the S3 object URL of the artifact in the bucket.
- The core device's token exchange role doesn't allow the AWS IoT Greengrass Core software to download the component artifact from the S3 object URL that you specify in the component's

recipe. Check that the token exchange role allows `s3:GetObject` for the S3 object URL where the artifact is available.

software.amazon.awssdk.services.s3.model.S3Exception: Access Denied (Service: S3, Status Code: 403, Request ID: <requestID>

The PackageDownloadException error might include this stack trace when the core device doesn't have permission to call `s3:GetBucketLocation`. The error message also includes the following message.

```
reason: Failed to determine S3 bucket location
```

Check that the core device's token exchange role allows `s3:GetBucketLocation` for the S3 bucket where the artifact is available.

Error: com.aws.greengrass.componentmanager.exceptions.ArtifactChecksumMisma Integrity check for downloaded artifact failed. Probably due to file corruption.

You might see this error when the AWS IoT Greengrass Core software fails to download a component artifact when the core device applies a deployment. The deployment fails because the downloaded artifact file's checksum doesn't match the checksum that AWS IoT Greengrass calculated when you created the component.

Do the following:

- Check if the artifact file changed in the S3 bucket where you host it. If the file changed since you created the component, restore it to the previous version that the core device expects. If you can't restore the file to its previous version, or if you want to use the new version of the file, create a new version of the component with the artifact file.
- Check your core device's internet connection. This error can occur if the artifact file becomes corrupted while it downloads. Create a new deployment and try again.

Error: com.aws.greengrass.componentmanager.exceptions.NoAvailableComponent Failed to negotiate component <name> version with cloud and no local applicable version satisfying requirement <requirements>

You might see this error when a core device can't find a component version that meets the requirements of the deployments for that core device. The core device checks for the component in the AWS IoT Greengrass service and on the local device. The error message includes each deployment's target and that deployment's version requirements for the component. The deployment target can be a thing, a thing group, or `LOCAL_DEPLOYMENT`, which represents the local deployment on the core device.

This issue can occur in the following cases:

- The core device is the target of multiple deployments that have conflicting component version requirements. For example, the core device might be the target of multiple deployments that include a `com.example.HelloWorld` component, where one deployment requires version 1.0.0 and the other requires version 1.0.1. It's impossible to have a component that meets both requirements, so the deployment fails.
- The component version doesn't exist in the AWS IoT Greengrass service or on the local device. The component might have been deleted, for example.
- There exists component versions that meet the version requirements, but none are compatible with the core device's platform.
- The core device's AWS IoT policy doesn't grant the `greengrass:ResolveComponentCandidates` permission. Look for `Status Code: 403` in the error log to identify this issue. To resolve this issue, add the `greengrass:ResolveComponentCandidates` permission to the core device's AWS IoT policy. For more information, see Minimal AWS IoT policy for AWS IoT Greengrass V2 core devices.

To resolve this issue, revise the deployments to include compatible component versions or remove incompatible ones. For more information about how to revise cloud deployments, see Revise deployments. For more information about how to revise local deployments, see the AWS IoT Greengrass CLI deployment create command.

software.amazon.awssdk.services.greengrassv2data.model.ResourceNotFoun The latest version of Component <componentName> doesn't claim platform <coreDevicePlatform> compatibility

You might see this error when you deploy a component to a core device, and the component doesn't list a platform that is compatible with the core device's platform. Do one of the following:

- If the component is a custom Greengrass component, you can update the component to be compatible with the core device. Add a new manifest that matches the core device's platform, or update an existing manifest to match the core device's platform. For more information, see AWS IoT Greengrass component recipe reference.
- If the component is provided by AWS, check if another version of the component is compatible with the core device. If no version is compatible, contact us on AWS re:Post using the AWS IoT Greengrass tag, or contact Support.

com.aws.greengrass.componentmanager.exceptions.PackagingException: The deployment attempts to update the nucleus from aws.greengrass.Nucleus-<version> to aws.greengrass.Nucleus-<version> but no component of type nucleus was included as target component

You might see this error when you deploy a component that depends on the Greengrass nucleus, and the core device runs an earlier Greengrass nucleus version than the latest minor version available. This error occurs because the AWS IoT Greengrass Core software tries to automatically update components to the latest compatible version. However, the AWS IoT Greengrass Core software prevents the Greengrass nucleus from updating to a new minor version, because several AWS-provided components depend on specific minor versions of the Greengrass nucleus. For more information, see Greengrass nucleus update behavior.

You must revise the deployment to specify the Greengrass nucleus version that you want to use. Do one of the following:

- Revise the deployment to specify the Greengrass nucleus version that the core device currently runs.

- Revise the deployment to specify a later minor version of the Greengrass nucleus. If you choose this option, you must also update the versions of all AWS-provided components that depend on specific minor versions of the Greengrass nucleus. For more information, see AWS-provided components.

Error: com.aws.greengrass.deployment.exceptions.DeploymentException: Unable to process deployment. Greengrass launch directory is not set up or Greengrass is not set up as a system service

You might see this error when you move a Greengrass device from one thing group to another, and then back to the original group with deployments that require Greengrass to restart.

To resolve this issue, recreate the launch directory for the device. We also strongly recommend upgrading to version 2.9.6 or later of the Greengrass nucleus.

The following is a Linux script to recreate the launch directory. Save the script in a file called `fix_directory.sh`.

```
#!/bin/bash

set -e

GG_ROOT=$1
GG_VERSION=$2

CURRENT="$GG_ROOT/alts/current"

if [ ! -L "$CURRENT" ]; then
  mkdir -p $GG_ROOT/alts/directory_fix
  echo "Relinking $GG_ROOT/alts/directory_fix to $CURRENT"
  ln -sf $GG_ROOT/alts/directory_fix $CURRENT
fi

TARGET=$(readlink $CURRENT)

if [[ ! -d "$TARGET" ]]; then
  echo "Creating directory: $TARGET"
  mkdir -p "$TARGET"
fi
```

```
DISTRO_LINK="$TARGET/distro"
DISTRO="$GG_ROOT/packages/artifacts-unarchived/aws.greengrass.Nucleus/$GG_VERSION/
aws.greengrass.nucleus/"
echo "Relinking Nucleus artifacts to $DISTRO_LINK"
ln -sf $DISTRO $DISTRO_LINK
```

To run the script, execute the following command:

```
[root@ip-172-31-27-165 ~]# ./fix_directory.sh /greengrass/v2 2.9.5
Relinking /greengrass/v2/alts/directory_fix to /greengrass/v2/alts/current
Relinking Nucleus artifacts to /greengrass/v2/alts/directory_fix/distro
```

Info: com.aws.greengrass.deployment.exceptions.RetryableDeploymentDocument Greengrass Cloud Service returned an error when getting full deployment configuration

You might see this error when the core device receives a large deployment document, which is a deployment document larger than 7 KB (for deployments that target things) or 31 KB (for deployments that target thing groups). To retrieve a large deployment document, a core device's AWS IoT policy must allow the `greengrass:GetDeploymentConfiguration` permission. This error can occur when the core device doesn't have this permission. When this error occurs, the deployment retries indefinitely, and its status is **In progress** (`IN_PROGRESS`).

To resolve this issue, add the `greengrass:GetDeploymentConfiguration` permission to the core device's AWS IoT policy. For more information, see Update a core device's AWS IoT policy.

Warn: com.aws.greengrass.deployment.DeploymentService: Failed to get thing group hierarchy

You might see this warning when the core device receives a deployment and the core device's AWS IoT policy doesn't allow the `greengrass:ListThingGroupsForCoreDevice` permission. When you create a deployment, the core device uses this permission to identify its thing groups and remove components for any thing groups from which you removed the core device. If the core device runs Greengrass nucleus v2.5.0, the deployment fails. If the core device runs Greengrass nucleus v2.5.1 or later, the deployment proceeds but doesn't remove components. For more

information about thing group removal behavior, see Deploy AWS IoT Greengrass components to devices.

To update the core device's behavior to remove components for thing groups from which you remove the core device, add the `greengrass:ListThingGroupsForCoreDevice` permission to the core device's AWS IoT policy. For more information, see Update a core device's AWS IoT policy.

Deployment unexpectedly removes components from a core device

A deployment removes components from the core device that you didn't target for removal, and the device stops running them.

On Greengrass nucleus v2.18.2 and earlier, a specific sequence of events causes a device to lose its record of which components each thing group deployed. After this loss, a deployment treats those components as orphaned and removes them. Greengrass nucleus v2.18.3 fixes this issue.

The sequence begins when the first deployment to a device fails and rolls back. For the full sequence, see How a device loses this record.

How to check whether a device has lost this record

These steps check the device's configuration to determine the cause of the removal. They don't identify which components a deployment removed. Inspect */greengrass/v2*`/config/effectiveConfig.yaml` on the core device.

1. To check whether the device's internal state is damaged, examine the `dependencies` list under the `main` service in `services`. Damaged internal state is the condition that leads to losing the record. On a device that has completed at least one successful deployment, this list includes DeploymentService, FleetStatusService, TelemetryAgent, and UpdateSystemPolicyService. If these names are absent on a device that has previously completed a successful deployment, the internal state is damaged. On a device that has never completed a successful deployment, a list containing only the nucleus is normal and doesn't indicate a problem.
2. To check whether the device has already lost the record, find DeploymentService under `services` and look for the GroupToRootComponents configuration. On a healthy device that has received at least one deployment, this configuration lists each thing group and the components that group deploys. If this configuration is missing or empty while components are installed and running on the device, the device has lost the record.

Whether deployments remove components after the device loses this record

After a device has lost this record, whether it loses components depends on which deployment runs next.

A deployment removes components when it runs while the record is missing and doesn't carry the missing group's component list:

- A local deployment runs. A local deployment carries only its own changes and relies on the device's local records to preserve everything else. Some fleets run a component that submits a local deployment every time the device starts. This is why the removal often appears immediately after a restart.
- On a device that belongs to more than one thing group, a deployment targets only some of those groups. That deployment recovers the groups it targets. It removes the components of the groups it doesn't target.

A deployment doesn't remove components when:

- No deployment runs on the device. The installed components continue to run normally. The device might remain in this state indefinitely with no impact.
- The next deployment targets the thing group whose component record the device is missing. That deployment carries the group's full component list and restores the record, which recovers the device automatically.

> **Important**
>
> Until you recover the device, avoid running local deployments on it. If a component on the device submits a local deployment when the device starts, recover the device before it restarts again. That deployment removes the components.

Recovery

Restarting the device doesn't restore the missing record. You must create deployments to recover the device.

To recover a device that lost this record, create a deployment that targets each thing group the core device belongs to. A deployment that targets a thing group carries the complete list of

components for that group. This rebuilds the missing local record. A local deployment can't recover the device because it doesn't carry other groups' component information.

> **Important**
>
> On a device that belongs to multiple thing groups, each deployment recovers only the targeted group. A deployment to one group might remove components belonging to groups you have not yet redeployed to. Create a deployment for every thing group the device belongs to.

After you create the recovery deployments, verify that each deployment completes successfully. If a recovery deployment fails and rolls back on a nucleus version earlier than 2.18.3, the device loses the record again. You must repeat the recovery.

After a recovery deployment completes successfully, the device doesn't lose this record again. This is true even if the device still runs Greengrass nucleus v2.18.2 or earlier. The deployment restores the internal state that protects the record, so later deployment failures and rollbacks don't cause this loss. If you reinstall the AWS IoT Greengrass Core software, the device might lose this record again.

Prevention

Upgrade the core device to Greengrass nucleus v2.18.3 or later to prevent this issue.

> **Note**
>
> Monitor nucleus upgrade deployments on devices that might have lost this record. If a nucleus upgrade deployment fails and rolls back on an unfixed device, the device might immediately lose this record. If this happens, create a deployment that targets each of the device's thing groups immediately after the failed upgrade to prevent component loss.

How a device loses this record

A device loses this record when all of the following conditions are true:

1. The device has not yet received a successful deployment.
2. The first deployment to the device fails after activation and rolls back.

3. A subsequent deployment runs on the device, which damages the internal state.
4. The device restarts at a later time.

A device whose first deployment completed successfully doesn't lose this record.

Info: com.aws.greengrass.deployment.DeploymentDocumentDownloader: Calling Greengrass cloud to get full deployment configuration

You might see this information message printed multiple times without an error, because the core device logs the error at the `DEBUG` log level. This issue can occur when the core device receives a large deployment document. When this issue occurs, the deployment retries indefinitely, and its status is **In progress** (`IN_PROGRESS`). For more information about how to resolve this issue, see this troubleshooting entry.

Caused by: software.amazon.awssdk.services.greengrassv2data.model.GreengrassV2Dat null (Service: GreengrassV2Data, Status Code: 403, Request ID: <some_request_id>, Extended Request ID: null)

You might see this error when a dataplane API doesn't have `iot:Connect` permission. If you don't have the correct policy, you'll receive a `GreengrassV2DataException: 403`. To create a permission policy, follow these instructions: Create an AWS IoT policy.

Core device component issues

Troubleshoot Greengrass component issues on core devices.

Topics

- Warn: '<command>' is not recognized as an internal or external command
- Python script doesn't log messages
- Component configuration doesn't update when changing default configuration
- awsiot.greengrasscoreipc.model.UnauthorizedError
- com.aws.greengrass.authorization.exceptions.AuthorizationException: Duplicate policy ID "<id>" for principal "<componentList>"

- com.aws.greengrass.tes.CredentialRequestHandler: Error in retrieving AwsCredentials from TES (HTTP 400)
- com.aws.greengrass.tes.CredentialRequestHandler: Error in retrieving AwsCredentials from TES (HTTP 403)
- com.aws.greengrass.tes.CredentialsProviderError: Could not load credentials from any providers
- Received error when attempting to retrieve ECS metadata: Could not connect to the endpoint URL: "<tokenExchangeServiceEndpoint>"
- copyFrom: <configurationPath> is already a container, not a leaf
- com.aws.greengrass.componentmanager.plugins.docker.exceptions.DockerLoginException: Error logging into the registry using credentials - 'The stub received bad data.'
- java.io.IOException: Cannot run program "cmd" ...: [LogonUser] The password for this account has expired.
- aws.greengrass.StreamManager: Instant exceeds minimum or maximum instant

Warn: '<command>' is not recognized as an internal or external command

You might see this error in a Greengrass component's logs when the AWS IoT Greengrass Core software fails to run a command in the component's lifecycle script. The component's state becomes `BROKEN` as a result of this error. This error can occur if the system user that runs the component, such as `ggc_user`, can't find the command's executable in the folders in the `PATH`.

On Windows devices, check that the folder that contains the executable is in the `PATH` for the system user that runs the component. If it's missing from the `PATH`, do one of the following:

- Add the executable's folder to the `PATH` system variable, which is available to all users. Then, restart the component.

  If you run Greengrass nucleus 2.5.0, after you update the `PATH` system variable, you must restart the AWS IoT Greengrass Core software to run components with the updated `PATH`. If the AWS IoT Greengrass Core software doesn't use the updated `PATH` after you restart the software, restart the device and try again. For more information, see Run the AWS IoT Greengrass Core software.
- Add the executable's folder to the `PATH` user variable for the system user that runs the component.

Python script doesn't log messages

Greengrass core devices collect logs that you can use to identify issues with components. If your Python script's `stdout` and `stderr` messages don't appear in your component logs, you might need to flush the buffer or disable buffering for these standard output streams in Python. Do any of the following:

- Run Python with the -u argument to disable buffering on `stdout` and `stderr`.

  Linux or Unix

  ```
  python3 -u hello_world.py
  ```

  Windows

  ```
  py -3 -u hello_world.py
  ```

- Use Setenv in your component's recipe to set the PYTHONUNBUFFERED environment variable to a non-empty string. This environment variable disables buffering on `stdout` and `stderr`.
- Flush the buffer for the `stdout` or `stderr` streams. Do one of the following:
  - Flush a message when you print.

    ```
    import sys

    print('Hello, error!', file=sys.stderr, flush=True)
    ```

  - Flush a message after you print. You can send multiple messages before you flush the stream.

    ```
    import sys

    print('Hello, error!', file=sys.stderr)
    sys.stderr.flush()
    ```

For more information about how to verify that your Python script outputs log messages, see Monitor AWS IoT Greengrass logs.

Component configuration doesn't update when changing default configuration

When you change the `DefaultConfiguration` in a component's recipe, the new default configuration won't replace the component's existing configuration during a deployment. To apply the new default configuration, you must reset the component's configuration to its default settings. When you deploy the component, specify a single empty string as the reset update.

Console

Reset paths

```
[""]
```

AWS CLI

The following command creates a deployment to a core device.

```
aws greengrassv2 create-deployment --cli-input-json file://reset-configuration-
deployment.json
```

The `reset-configuration-deployment.json` file contains the following JSON document.

```
{
  "targetArn": "arn:aws:iot:us-west-2:123456789012:thing/MyGreengrassCore",
  "deploymentName": "Deployment for MyGreengrassCore",
  "components": {
    "com.example.HelloWorld": {
      "componentVersion": "1.0.0",
      "configurationUpdate": {,
        "reset": [""]
      }
    }
  }
}
```

Greengrass CLI

The following Greengrass CLI command creates a local deployment on a core device.

```
sudo greengrass-cli deployment create \
  --recipeDir recipes \
  --artifactDir artifacts \
  --merge "com.example.HelloWorld=1.0.0" \
  --update-config reset-configuration-deployment.json
```

The `reset-configuration-deployment.json` file contains the following JSON document.

```
{
  "com.example.HelloWorld": {
    "RESET": [""]
  }
}
```

awsiot.greengrasscoreipc.model.UnauthorizedError

You might see this error in a Greengrass component's logs when the component doesn't have permission to perform an IPC operation on a resource. To grant a component permission to call an IPC operation, define an IPC authorization policy in the component's configuration. For more information, see Authorize components to perform IPC operations.

Tip

If you change the `DefaultConfiguration` in a component's recipe, you must reset the component's configuration to its new default configuration. When you deploy the component, specify a single empty string as the reset update. For more information, see Component configuration doesn't update when changing default configuration.

com.aws.greengrass.authorization.exceptions.AuthorizationException: Duplicate policy ID "<id>" for principal "<componentList>"

You might see this error if multiple IPC authorization policies, including across all components on the core device, use the same policy ID.

Check your components' IPC authorization policies, fix any duplicates, and try again. To create unique policy IDs, we recommend that you combine the component name, IPC service name, and a counter. For more information, see Authorize components to perform IPC operations.

Tip

If you change the `DefaultConfiguration` in a component's recipe, you must reset the component's configuration to its new default configuration. When you deploy the component, specify a single empty string as the reset update. For more information, see Component configuration doesn't update when changing default configuration.

com.aws.greengrass.tes.CredentialRequestHandler: Error in retrieving AwsCredentials from TES (HTTP 400)

You might see this error when a core device can't get AWS credentials from the token exchange service. The HTTP 400 status code indicates that this error occurred because the core device's token exchange IAM role doesn't exist or doesn't have a trust relationship that allows the AWS IoT credentials provider to assume it.

Do the following:

1. Identify the token exchange role that the core device uses. The error message includes the core device's AWS IoT role alias, which points to the token exchange role. Run the following command on your development computer, and replace *MyGreengrassCoreTokenExchangeRoleAlias* with the name of the AWS IoT role alias from the error message.

   ```
   aws iot describe-role-alias --role-alias MyGreengrassCoreTokenExchangeRoleAlias
   ```

   The response includes the Amazon Resource Name (ARN) of the token exchange IAM role.

   ```
   {
     "roleAliasDescription": {
       "roleAlias": "MyGreengrassCoreTokenExchangeRoleAlias",
       "roleAliasArn": "arn:aws:iot:us-west-2:123456789012:rolealias/
   MyGreengrassCoreTokenExchangeRoleAlias",
       "roleArn": "arn:aws:iam::123456789012:role/MyGreengrassV2TokenExchangeRole",
       "owner": "123456789012",
       "credentialDurationSeconds": 3600,
       "creationDate": "2021-02-05T16:46:18.042000-08:00",
       "lastModifiedDate": "2021-02-05T16:46:18.042000-08:00"
     }
   ```

```
}
```

2. Check that the role exists. Run the following command, and replace *MyGreengrassV2TokenExchangeRole* with the name of the token exchange role.

   ```
   aws iam get-role --role-name MyGreengrassV2TokenExchangeRole
   ```

   If the command returns a `NoSuchEntity` error, the role doesn't exist, and you must create it. For more information about how to create and configure this role, see Authorize core devices to interact with AWS services.

3. Check that the role has a trust relationship that allows the AWS IoT credentials provider to assume it. The response from the previous step contains an `AssumeRolePolicyDocument`, which defines the role's trust relationships. The role must define a trust relationship that allows `credentials.iot.amazonaws.com` to assume it. This document should look similar to the following example.

   JSON

   ```
   {
     "Version":"2012-10-17",
     "Statement": [
       {
         "Effect": "Allow",
         "Principal": {
           "Service": "credentials.iot.amazonaws.com"
         },
         "Action": "sts:AssumeRole"
       }
     ]
   }
   ```

   If the role's trust relationships don't allow `credentials.iot.amazonaws.com` to assume it, you must add this trust relationship to the role. For more information, see Modifying a role in the *AWS Identity and Access Management User Guide*.

com.aws.greengrass.tes.CredentialRequestHandler: Error in retrieving AwsCredentials from TES (HTTP 403)

You might see this error when a core device can't get AWS credentials from the token exchange service. The HTTP 403 status code indicates that this error occurred because the core device's AWS IoT policies don't grant the `iot:AssumeRoleWithCertificate` permission for the core device's AWS IoT role alias.

Review the core device's AWS IoT policies, and add the `iot:AssumeRoleWithCertificate` permission for the core device's AWS IoT role alias. The error message includes the core device's current AWS IoT role alias. For more information about this permission and how to update the core device's AWS IoT policies, see Minimal AWS IoT policy for AWS IoT Greengrass V2 core devices and Update a core device's AWS IoT policy.

com.aws.greengrass.tes.CredentialsProviderError: Could not load credentials from any providers

You might see this error when the component tries to request AWS credentials and can't connect to the token exchange service.

Do the following:

- Check that the component declares a dependency on the token exchange service component, `aws.greengrass.TokenExchangeService`. If it doesn't, add the dependency and redeploy the component.
- If the component runs in docker, ensure that you apply the right network settings and environment variables, according to Use AWS credentials in Docker container components (Linux).
- If the component is written in NodeJS, set dns.setDefaultResultOrder to **`ipv4first`**.
- Inspect `/etc/hosts` for an entry that starts with `::1` and contains `localhost`. Remove the entry to see if it caused the component to connect to the token exchange service at the wrong address.

Received error when attempting to retrieve ECS metadata: Could not connect to the endpoint URL: "<tokenExchangeServiceEndpoint>"

You might see this error when the component doesn't run the token exchange service and a component tries to request AWS credentials.

Do the following:

- Check that the component declares a dependency on the token exchange service component, `aws.greengrass.TokenExchangeService`. If it doesn't, add the dependency and redeploy the component.
- Check whether the component uses AWS credentials in its `install` lifecycle. AWS IoT Greengrass doesn't guarantee the availability of the token exchange service during the `install` lifecycle. Update the component to move the code that uses AWS credentials into the `startup` or `run` lifecycle, then redeploy the component.

copyFrom: <configurationPath> is already a container, not a leaf

You might see this error when you change a configuration value from a container type (a list or object) to a non-container type (a string, number, or Boolean). Do the following:

1. Check the component's recipe to see whether its default configuration sets that configuration value to a list or an object. If so, remove or change that configuration value.
2. Create a deployment to reset that configuration value to its default value. For more information, see Create deployments and Update component configurations.

Then, you can set that configuration value to a string, number, or Boolean.

com.aws.greengrass.componentmanager.plugins.docker.exceptions.DockerLo Error logging into the registry using credentials - 'The stub received bad data.'

You might see this error in the Greengrass nucleus logs when the Docker application manager component tries to download a Docker image from a private repository in Amazon Elastic Container Registry (Amazon ECR). This error occurs if you use the `wincred` Docker credential

helper (`docker-credential-wincred`). As a result, Amazon ECR is unable to store the login credentials.

Take one of the following actions:

- If you don't use the `wincred` Docker credential helper, remove the `docker-credential-wincred` program from the core device.
- If you use the `wincred` Docker credential helper, do the following:
  1. Rename the `docker-credential-wincred` program on the core device. Replace `wincred` with a new name for the Windows Docker credential helper. For example, you can rename it to `docker-credential-wincredreal`.
  2. Update the `credsStore` option in the Docker configuration file (`.docker/config.json`) to use the new name for the Windows Docker credential helper. For example, if you renamed the program to `docker-credential-wincredreal`, update the `credsStore` option to `wincredreal`.

     ```
     {
       "credsStore": "wincredreal"
     }
     ```

java.io.IOException: Cannot run program "cmd" ...: [LogonUser] The password for this account has expired.

You might see this error on a Windows core device when the system user that runs the component's processes, such as `ggc_user`, has an expired password. As a result, the AWS IoT Greengrass Core software is unable to run component processes as that system user.

To update a Greengrass system user's password

1. Run the following command as an administrator to set the user's password. Replace *`ggc_user`* with the system user, and replace *`password`* with the password to set.

   ```
   net user ggc_user password
   ```

2. Use the PsExec utility to store the user's new password in the Credential Manager instance for the LocalSystem account. Replace *`password`* with the user's password that you set.

```
psexec -s cmd /c cmdkey /generic:ggc_user /user:ggc_user /pass:password
```

Tip

Depending on your Windows configuration, the user's password might be set to expire at a date in the future. To ensure your Greengrass applications continue to operate, track when the password expires, and update it before it expires. You can also set the user's password to never expire.

- To check when a user and its password expire, run the following command.

  ```
  net user ggc_user | findstr /C:expires
  ```

- To set a user's password to never expire, run the following command.

  ```
  wmic UserAccount where "Name='ggc_user'" set PasswordExpires=False
  ```

- If you're using Windows 10 or later where the wmic command is deprecated, run the following PowerShell command.

  ```
  Get-CimInstance -Query "SELECT * from Win32_UserAccount WHERE name =
   'ggc_user'" | Set-CimInstance -Property @{PasswordExpires="False"}
  ```

aws.greengrass.StreamManager: Instant exceeds minimum or maximum instant

When you upgrade stream manager v2.0.7 to a version between v2.0.8 and v2.0.11, you might see the following error in the stream manager component's logs if the component fails to start.

```
2021-07-16T00:54:58.568Z [INFO] (Copier) aws.greengrass.StreamManager:
 stdout. Caused by: com.fasterxml.jackson.databind.JsonMappingException:
 Instant exceeds minimum or maximum instant (through reference chain:
 com.amazonaws.iot.greengrass.streammanager.export.PersistedSuccessExportStatesV1["lastExportTi
 {scriptName=services.aws.greengrass.StreamManager.lifecycle.startup.script,
 serviceName=aws.greengrass.StreamManager, currentState=STARTING}
```

```
2021-07-16T00:54:58.579Z [INFO] (Copier) aws.greengrass.StreamManager: stdout.
 Caused by: java.time.DateTimeException: Instant exceeds minimum or maximum instant.
 {scriptName=services.aws.greengrass.StreamManager.lifecycle.startup.script,
 serviceName=aws.greengrass.StreamManager, currentState=STARTING}
```

If you deployed stream manager v2.0.7 and you want to upgrade to a later version, you must upgrade to stream manager v2.0.12 directly. For more information about the stream manager component, see Stream manager.

Core device Lambda function component issues

Troubleshoot Lambda function component issues on core devices.

Topics

- The following cgroup subsystems are not mounted: devices, memory
- ipc_client.py:64,HTTP Error 400:Bad Request, b'No subscription exists for the source <label-or-lambda-arn> and subject <label-or-lambda-arn>

The following cgroup subsystems are not mounted: devices, memory

You might see this error when you run a containerized Lambda function in the following cases:

- The core device doesn't have cgroup v1 enabled for the memory or device cgroups.
- The core device has cgroups v2 enabled. Greengrass Lambda functions require cgroups v1, and cgroups v1 and v2 are mutually exclusive.

To enable cgroups v1, boot the device with the following Linux kernel parameters.

```
cgroup_enable=memory cgroup_memory=1 systemd.unified_cgroup_hierarchy=0
```

> **Tip**
>
> On a Raspberry Pi, edit the `/boot/cmdline.txt` file to set the device's kernel parameters.

ipc_client.py:64,HTTP Error 400:Bad Request, b'No subscription exists for the source <label-or-lambda-arn> and subject <label-or-lambda-arn>

You might see this error when you run a V1 Lambda function, which uses the AWS IoT Greengrass Core SDK, on a V2 core device without specifying a subscription in the legacy subscription router component. To fix this issue, deploy and configure the legacy subscription router to specify the required subscriptions. For more information, see Import V1 Lambda functions.

Component version discontinued

You might see a notification on your Personal Health Dashboard (PHD) when a component version on your core device is discontinued. The component version sends this notification to your PHD within 60 minutes of being discontinued.

To see which deployments you need to revise, do the following using the AWS Command Line Interface:

1. Run the following command to get a list of your core devices.

   ```
   aws greengrassv2 list-core-devices
   ```

2. Run the following command to retrieve the status of the components on each core device from Step 1. Replace *coreDeviceName* with the name of each core device to query.

   ```
   aws greengrassv2 list-installed-components --core-device-thing-name coreDeviceName
   ```

3. Gather the core devices with the discontinued component version installed from the previous steps.
4. Run the following command to retrieve the status of all deployment jobs for each core device from Step 3. Replace *coreDeviceName* with the name of the core device to query.

   ```
   aws greengrassv2 list-effective-deployments --core-device-thing-name coreDeviceName
   ```

   The response contains the list of deployment jobs for the core device. You can revise the deployment to choose another component version. For more information about how to revise a deployment, see Revise deployments.

Greengrass Command Line Interface issues

Troubleshoot issues with the Greengrass CLI.

Topics

- java.lang.RuntimeException: Unable to create ipc client

java.lang.RuntimeException: Unable to create ipc client

You might see this error when you run a Greengrass CLI command and you specify a different root folder than where the AWS IoT Greengrass Core software is installed.

Do one of the following to set the root path, and replace */greengrass/v2* with the path to your AWS IoT Greengrass Core software installation:

- Set the `GGC_ROOT_PATH` environment variable to */greengrass/v2*.
- Add the `--ggcRootPath` */greengrass/v2* argument to your command as shown in the following example.

```
greengrass-cli --ggcRootPath /greengrass/v2 <command> <subcommand> [arguments]
```

AWS Command Line Interface issues

Troubleshoot AWS CLI issues for AWS IoT Greengrass V2.

Topics

- Error: Invalid choice: 'greengrassv2'

Error: Invalid choice: 'greengrassv2'

You might see this error when you run an AWS IoT Greengrass V2 command with the AWS CLI (for example, `aws greengrassv2 list-core-devices`).

This error indicates that you have a version of the AWS CLI that doesn't support AWS IoT Greengrass V2. To use AWS IoT Greengrass V2 with the AWS CLI, you must have one of the following versions or later:

- Minimum AWS CLI V1 version: v1.18.197
- Minimum AWS CLI V2 version: v2.1.11

Tip

You can run the following command to check the version of the AWS CLI that you have.

```
aws --version
```

To resolve this issue, update the AWS CLI to a later version that supports AWS IoT Greengrass V2. For more information, see Installing, updating, and uninstalling the AWS CLI in the *AWS Command Line Interface User Guide*.

Detailed deployment error codes

Use the error codes and solutions in these sections to help resolve issues with component deployment when using the Greengrass nucleus version 2.8.0 or later.

The Greengrass nucleus reports deployment errors as a hierarchy from least specific to the most specific code available. You can use this hierarchy to help pinpoint the reason for a deployment error. For example, the following is a possible error hierarchy:

- DEPLOYMENT_FAILURE
  - ARTIFACT_DOWNLOAD_ERROR
    - IO_ERROR
      - DISK_SPACE_CRITICAL

The error codes are organized into types. Each type represents a class of errors that can occur. AWS IoT Greengrass reports these errors types in the console, the API, and AWS CLI. There can be more than one error type, depending on the errors reported in the error hierarchy. For the preceding example, the error type returned is `DEVICE_ERROR`.

The types are:

- **PERMISSION_ERROR** – Access to an operation that requires permission was denied.

- **REQUEST_ERROR** – An error occurred due to an issue in the deployment document.
- **COMPONENT_RECIPE_ERROR** – An error occurred due to an issue in a component recipe.
- **AWS_COMPONENT_ERROR** – An error occurred when starting or removing an AWS provided component.
- **USER_COMPONENT_ERROR** – An error occurred when starting or removing a user component.
- **COMPONENT_ERROR** – An error occurred when starting or removing a component, but the Greengrass nucleus couldn't determine if the component is an AWS provided component or a user component.
- **DEVICE_ERROR** – An error occurred with local I/O or another device error occurred.
- **DEPENDENCY_ERROR** – A deployment failed to download an artifact from Amazon S3 or to pull an image from an ECR registry.
- **HTTP_ERROR** – An error occurred with an HTTP request.
- **NETWORK_ERROR** – An error occurred with the device network.
- **NUCLEUS_ERROR** – The Greengrass nucleus could not locate a component or could not find the active nucleus version.
- **SERVER_ERROR** – A server returned a 500 error in response to a request.
- **CLOUD_SERVICE_ERROR** – An error occurred with the AWS IoT Greengrass cloud service.
- **UNKNOWN_ERROR** – An unchecked exception was thrown by the component.

Many of the errors in this section report additional information in the AWS IoT Greengrass Core logs. These logs are stored on the core device's local file system. There are logs for the AWS IoT Greengrass Core core software and for each individual component. For information on accessing the logs, see Access file system logs.

Permission error

ACCESS_DENIED

You might get this error when an AWS service operation returns a 403 error because permissions are not set up correctly. Check the more specific error code for details.

GET_DEPLOYMENT_CONFIGURATION_ACCESS_DENIED

You might get this error when the AWS IoT policy doesn't allow permission to call the `GetDeploymentConfiguration` operation. Add the `greengrass::GetDeploymentConfiguration` permission to the core device's policy.

GET_COMPONENT_VERSION_ARTIFACT_ACCESS_DENIED

You might get this error when the core device AWS IoT policy doesn't allow the `greengrass:GetComponentVersionArtifact` permission. Add the permission to the core device's policy.

RESOLVE_COMPONENT_CANDIDATES_ACCESS_DENIED

You might get this error when the core device AWS IoT policy doesn't allow the `greengrass:ResolveComponentCandidates` permission. Add the permission to the core device's policy.

GET_ECR_CREDENTIAL_ERROR

You might get this error when the deployment couldn't authenticate with a private registry in ECR. Check the log for a specific error and then try the deployment again.

USER_NOT_AUTHORIZED_FOR_DOCKER

You might get this error when the Greengrass user is not authorized to use Docker. Make sure that you are running Greengrass as root or that the user is added to the `docker` group. Then try the deployment again.

S3_ACCESS_DENIED

You might get this error when an Amazon S3 operation returns a 403 error. Check any additional error codes or logs for details.

S3_HEAD_OBJECT_ACCESS_DENIED

You might get this error either when the device's token exchange role doesn't allow the AWS IoT Greengrass Core software to download the component artifact from the S3 object URL that you specify in the component's recipe or that the component artifact isn't available. Check that the token exchange role allows `s3:GetObject` for the S3 object URL where the artifact is available and that the artifact is present.

S3_GET_BUCKET_LOCATION_ACCESS_DENIED

You might get this error when the device's token exchange role doesn't allow the `s3:GetBucketLocation` permission for the Amazon S3 bucket where the artifact is available. Check that the device's allows the permission then try the deployment again.

S3_GET_OBJECT_ACCESS_DENIED

You might get this error either when the device's token exchange role doesn't allow the AWS IoT Greengrass Core software to download the component artifact from the S3 object URL that you

specify in the component's recipe or that the component artifact isn't available. Check that the token exchange role allows `s3:GetObject` for the S3 object URL where the artifact is available and that the artifact is present.

MISSING_MQTT_CONNECT_POLICY

You might get this error when the AWS IoT policy doesn't allow `iot:Connect` for the client ID that the nucleus uses during the pre-flight connectivity check for an IoT connectivity switch deployment. The pre-flight check uses client ID *`thing-name`*`#endpoint-switch`. Update the `iot:Connect` resource in the AWS IoT policy to `arn:aws:iot:*:*:client/`*`thing-name`*`*` to allow both the standard and pre-flight client IDs.

CREDENTIAL_ENDPOINT_AUTH_FAILURE

You might get this error when the AWS IoT Credential Provider returns a 401, 403, or 404 response during the pre-flight credential endpoint check for an IoT connectivity switch deployment. Verify that the role alias exists in the destination Region, that you authorized the device certificate for the role alias, and that the AWS IoT policy includes `iot:AssumeRoleWithCertificate` permission.

Request error

NUCLEUS_MISSING_REQUIRED_CAPABILITIES

You might get this error when the nucleus version in the deployment isn't capable a requested operation, such as downloading a large configuration or setting Linux resource limits. Retry the deployment with a nucleus version that supports the operation.

MULTIPLE_NUCLEUS_RESOLVED_ERROR

You might get this error when a deployment attempts to deploy multiple nucleus components. Check the log to see what caused the error, then check the nucleus software update page to see if the issue has been corrected in a later version of the nucleus, or contact Support.

COMPONENT_CIRCULAR_DEPENDENCY_ERROR

You might get this error when two components in your deployment depend on each other. Revise the component setup so that the components in your deployment don't rely on each other.

UNAUTHORIZED_NUCLEUS_MINOR_VERSION_UPDATE

You might get this error when a component in your deployment requires a nucleus minor version update, but that version isn't specified in the deployment. This helps to reduce accidental minor version updates for components that depend on a different version. Include the new minor nucleus version in the deployment.

MISSING_DOCKER_APPLICATION_MANAGER

You might get this error when you deploy a Docker component without deploying the Docker application manager. Make sure that your deployment includes the Docker application manager.

MISSING_TOKEN_EXCHANGE_SERVICE

You might get this error when the deployment wants to download a Docker image artifact from a private ECR registry without deploying the token exchange service. Make sure that your deployment includes the token exchange service.

COMPONENT_VERSION_REQUIREMENTS_NOT_MET

You might get this error when there is a version constraint conflict or a component version does not exist. For more information, see Error: com.aws.greengrass.componentmanager.exceptions.NoAvailableComponentVersionException: Failed to negotiate component <name> version with cloud and no local applicable version satisfying requirement <requirements>.

THROTTLING_ERROR

You might get this error when an AWS service operation exceeded a rate quota. Retry the deployment.

CONFLICTED_REQUEST

You might get this error when an AWS service operation returns a 409 error because you deployment is trying to perform more than one operation at a time. Retry the deployment.

RESOURCE_NOT_FOUND

You might get this error when an AWS service operation returns a 404 error because a resource couldn't be found. Check the log for the missing resource.

RUN_WITH_CONFIG_NOT_VALID

You might get this error when the `posixUser`, `posixGroup`, or `windowsUser` information specified to run the component isn't valid. Check that the user is valid and then retry the deployment.

UNSUPPORTED_REGION

You might get this error when the Region specified for the deployment isn't supported by AWS IoT Greengrass. Check the Region and try the deployment again.

IOT_CRED_ENDPOINT_NOT_VALID

You might get this error when the AWS IoT credential endpoint specified in the configuration isn't valid. Check the endpoint and try your request again.

IOT_DATA_ENDPOINT_NOT_VALID

You might get this error when the AWS IoT data endpoint specified in the configuration isn't valid. Check the endpoint and try your request again.

S3_HEAD_OBJECT_RESOURCE_NOT_FOUND

You might get this error when the component artifact isn't available at the S3 object URL that you specify in the component's recipe. Check that you uploaded the artifact to the S3 bucket and that the artifact URI matches the S3 object URL of the artifact in the bucket.

S3_GET_BUCKET_LOCATION_RESOURCE_NOT_FOUND

You might get this error when the Amazon S3 bucket isn't found. Check that the bucket exists and try the deployment again.

S3_GET_OBJECT_RESOURCE_NOT_FOUND

You might get this error when the component artifact isn't available at the S3 object URL that you specify in the component's recipe. Check that you uploaded the artifact to the S3 bucket and that the artifact URI matches the S3 object URL of the artifact in the bucket.

IO_MAPPING_ERROR

You might get this error when an I/O error occurs when parsing deployment document or recipe. Check any additional error codes or logs for details.

Component recipe error

RECIPE_PARSE_ERROR

You might get this error when the deployment recipe couldn't be parsed because there is an error in the structure of the recipe. Check that the recipe is correctly formatted and try the deployment again.

RECIPE_METADATA_PARSE_ERROR

You might get this error when the deployment recipe metadata downloaded from the cloud couldn't be parsed. Contact Support.

ARTIFACT_URI_NOT_VALID

You might get this error when an artifact URI in a recipe isn't formatted correctly. Check the log for the URI that isn't valid, update the URI in the recipe, then try the deployment again.

S3_ARTIFACT_URI_NOT_VALID

You might get this error when the Amazon S3 URI of an artifact in a recipe isn't valid. Check the log for the URI that isn't valid, update the URI in the recipe, then try the deployment again.

DOCKER_ARTIFACT_URI_NOT_VALID

You might get this error when the Docker URI of an artifact in a recipe isn't valid. Check the log for the URI that isn't valid, update the URI in the recipe, then try the deployment again.

EMPTY_ARTIFACT_URI

You might get this error when the URI of an artifact is not specified in a recipe. Check the log for the artifact that is missing a URI, update the URI in the recipe, then try the deployment again.

EMPTY_ARTIFACT_SCHEME

You might get this error when a URI scheme is not defined for an artifact. Check the log for the URI that isn't valid, update the URI in the recipe, then try the deployment again.

UNSUPPORTED_ARTIFACT_SCHEME

You might get this error when a URI scheme isn't supported by the running nucleus version. Either a URI isn't valid or you need to update the nucleus version. If the URI isn't valid, check the log for the URI that isn't valid, update the URI in the recipe, then try the deployment again.

RECIPE_MISSING_MANIFEST

You might get this error when the manifest section isn't included in the recipe. Add the manifest to the recipe and try the deployment again.

RECIPE_MISSING_ARTIFACT_HASH_ALGORITHM

You might get this error when an artifact that is not local is specified inside a recipe without a hash algorithm. Add the algorithm to the artifact and then try the request again.

ARTIFACT_CHECKSUM_MISMATCH

You might get this error when a downloaded artifact has a different digest than the one specified in the recipe. Make sure that the recipe contains the correct digest and then try the deployment again. For more information see Error: com.aws.greengrass.componentmanager.exceptions.ArtifactChecksumMismatchException: Integrity check for downloaded artifact failed. Probably due to file corruption..

COMPONENT_DEPENDENCY_NOT_VALID

You might get this error when the dependency type specified in a deployment recipe isn't valid. Check the recipe and then try your request again.

CONFIG_INTERPOLATE_ERROR

You might get this error when interpolating a recipe variable. Check the log for details.

IO_MAPPING_ERROR

You might get this error when an I/O error occurs when parsing deployment document or recipe. Check any additional error codes or logs for details.

AWS component error, user component error, component error

The following error codes are returned when there is a problem with a component. The actual error type reported depends on the specific component that raised the error. If the Greengrass nucleus identifies the component as one provided by AWS IoT Greengrass, it returns `AWS_COMPONENT_ERROR`. If the component is identified as a user component, the Greengrass nucleus returns `USER_COMPONENT_ERROR`. If the Greengrass nucleus can't tell, it returns `COMPONENT_ERROR`.

COMPONENT_UPDATE_ERROR

You might get this error when a component doesn't update during a deployment. Check any additional error codes or check the log to see what caused the error.

COMPONENT_BROKEN

You might get this error when a component is broken during a deployment. Check the component log for error details and then try the deployment again.

REMOVE_COMPONENT_ERROR

You might get this error when the nucleus can't remove a component during a deployment. Check the log for error details and then try the deployment again.

COMPONENT_BOOTSTRAP_TIMEOUT

You might get this error when a component's bootstrap task took longer than the configured timeout. Increase the timeout or reduce the execution time of the bootstrap task, then try the deployment again.

COMPONENT_BOOTSTRAP_ERROR

You might get this error when a component's bootstrap task has an error. Check the log for error details, then try the deployment again.

COMPONENT_CONFIGURATION_NOT_VALID

You might get this error when the nucleus can't validate the deployed configuration for the component. Check the log for error details, then try the deployment again.

Device error

IO_WRITE_ERROR

You might get this error when writing to a file. Check the log for details.

IO_READ_ERROR

You might get this error when reading from a file. Check the log for details.

DISK_SPACE_CRITICAL

You might get this error when there is not enough disk space to complete a deployment request. You must have at least 20 Mb of available space, or enough to hold a larger artifact. Free up some disk space and then retry the deployment.

IO_FILE_ATTRIBUTE_ERROR

You might get this error when the existing file size can't be retrieved from the file system. Check the log for details.

SET_PERMISSION_ERROR

You might get this error when the permissions can't be set on a downloaded artifact or artifact directory. Check the log for details.

IO_UNZIP_ERROR

You might get this error when an artifact can't be unzipped. Check the log for details.

LOCAL_RECIPE_NOT_FOUND

You might get this error when the local copy of a recipe file couldn't be found. Try the deployment again.

LOCAL_RECIPE_CORRUPTED

You might get this error when the local copy of the recipe has changed since it was downloaded. Delete the existing copy of the recipe and try the deployment again.

LOCAL_RECIPE_METADATA_NOT_FOUND

You might get this error when the local copy of the recipe metadata file couldn't be found. Try the deployment again.

LAUNCH_DIRECTORY_CORRUPTED

You might get this error when the directory used to launch the Greengrass nucleus (`/greengrass/v2/alts/current`) has been modified since the last time the nucleus was started. Restart the nucleus and then retry the deployment.

HASHING_ALGORITHM_UNAVAILABLE

You might get this error when the device's Java distribution doesn't support the required hashing algorithm or when the hash algorithm specified in a component recipe isn't valid.

DEVICE_CONFIG_NOT_VALID_FOR_ARTIFACT_DOWNLOAD

You might get this error when there is an error in the device configuration that prevented the deployment from downloading the artifact from Amazon S3 or the Greengrass cloud. Check the log for a specific configuration error and then retry the deployment.

Dependency error

DOCKER_ERROR

You might get this error when pulling a Docker image. Check any additional error codes or logs for details.

DOCKER_SERVICE_UNAVAILABLE

You might get this error when Greengrass couldn't log into the Docker registry. Check the log for a specific error and then try the deployment again.

DOCKER_LOGIN_ERROR

You might get this error when an unexpected error occurs when logging in to Docker. Check the log for a specific error and then try the deployment again.

DOCKER_PULL_ERROR

You might get this error when an unexpected error occurs when pulling a Docker image from the registry. Check the log for a specific error and then try the deployment again.

DOCKER_IMAGE_NOT_VALID

You might get this error when the requested Docker image doesn't exist. Check the log for a specific error and try the deployment again.

DOCKER_IMAGE_QUERY_ERROR

You might get this error when an unexpected failure occurs when querying Docker for available images. Check the log for the specific error and try the deployment again.

S3_ERROR

You might get this error when downloading an Amazon S3 artifact. Check any additional error codes or logs for details.

S3_RESOURCE_NOT_FOUND

You might get this error when an Amazon S3 operation returns a 404 error. Check any additional error codes or logs for details.

S3_BAD_REQUEST

You might get this error when an Amazon S3 operation returns a 400 error. Check the log for a specific error and try the request again.

HTTP error

HTTP_REQUEST_ERROR

You might get this error when an error occurred when making an HTTP request. Check the log for the specific error.

DOWNLOAD_DEPLOYMENT_DOCUMENT_ERROR

You might get this error when an HTTP error occurred when downloading the deployment document. Check the log for the specific HTTP error.

GET_GREENGRASS_ARTIFACT_SIZE_ERROR

You might get this error when an HTTP error occurred when getting the size of a public component artifact. Check the log for the specific HTTP error.

DOWNLOAD_GREENGRASS_ARTIFACT_ERROR

You might get this error when an HTTP error occurred when downloading a public component artifact. Check the log for the specific HTTP error.

Network error

NETWORK_ERROR

You might get this error when there is a connection issue during a deployment. Check the connection of the device to the Internet and try the deployment again.

MQTT_CONNECTION_FAILED

You might get this error when the pre-flight MQTT connection to the new AWS IoT data endpoint fails during an IoT connectivity switch deployment. Check that the endpoint is correct, the device has network connectivity to the endpoint, and no firewall rules block MQTT traffic on the configured port.

TLS_HANDSHAKE_FAILURE

You might get this error when the TLS handshake fails during the pre-flight connectivity check to the new AWS IoT data endpoint. Verify that you registered the device certificate in the destination account and that the root CA certificate on the device trusts the endpoint.

Nucleus error

BAD_REQUEST

You might get this error when an AWS cloud operation returns a 400 error. Check the log to see which API caused the error, then check the nucleus software update page to see if the issue has been corrected in a later version of the nucleus, or contact Support.

NUCLEUS_VERSION_NOT_FOUND

You might get this error when a core device can't find the version of the active nucleus. Check the log to see what caused the error, then check the nucleus software update page to see if the issue has been corrected in a later version of the nucleus, or contact Support.

NUCLEUS_RESTART_FAILURE

You might get this error when the nucleus doesn't restart during any deployment that requires a nucleus restart. Check the loader log to see what caused the error, then check the nucleus software update page to see if the issue has been corrected in a later version of the nucleus, or contact Support.

INSTALLED_COMPONENT_NOT_FOUND

You might get this error when the nucleus can't locate an installed component. Check the log to see what caused the error, then check the nucleus software update page to see if the issue has been corrected in a later version of the nucleus, or contact Support.

DEPLOYMENT_DOCUMENT_NOT_VALID

You might get this error when the device receives deployment document that isn't valid. Check any additional error codes or check the log to see what caused the error.

EMPTY_DEPLOYMENT_REQUEST

You might get this error when a device receives an empty deployment request. Check the log to see what caused the error, then check the nucleus software update page to see if the issue has been corrected in a later version of the nucleus, or contact Support.

DEPLOYMENT_DOCUMENT_PARSE_ERROR

You might get this error when the deployment request format doesn't match the expected format. Check the log to see what caused the error, then check the nucleus software update page to see if the issue has been corrected in a later version of the nucleus, or contact Support.

COMPONENT_METADATA_NOT_VALID_IN_DEPLOYMENT

You might get this error when the deployment request contains component metadata that isn't valid. Check the log to see what caused the error, then check the nucleus software update page to see if the issue has been corrected in a later version of the nucleus, or contact Support.

LAUNCH_DIRECTORY_CORRUPTED

You might get this error when you move a Greengrass device from one thing group to another, and then back to the original group with deployments that require Greengrass to restart. To resolve the error, recreate the launch directory for Greengrass on the device.

For more information, see Error: com.aws.greengrass.deployment.exceptions.DeploymentException: Unable to process deployment. Greengrass launch directory is not set up or Greengrass is not set up as a system service.

Server error

SERVER_ERROR

You might get this error when an AWS service operation returns a 500 error because the service can't process the request right now. Retry the deployment later.

S3_SERVER_ERROR

You might get this error when an Amazon S3 operation returns a 500 error. Check any additional error codes or logs for details.

CREDENTIAL_ENDPOINT_SERVER_ERROR

You might get this error when the AWS IoT Credential Provider returns a server error or is unreachable after all retries during the pre-flight credential endpoint check for an IoT connectivity switch deployment. This typically indicates a transient service issue. Retry the deployment. If the error persists, check the AWS IoT Credential Provider service health in the destination Region.

Cloud service error

RESOLVE_COMPONENT_CANDIDATES_BAD_RESPONSE

You might get this error when the Greengrass cloud service sends an incompatible response to the `ResolveComponentCandidates` operation. Check the log to see what caused the error, then check the nucleus software update page to see if the issue has been corrected in a later version of the nucleus, or contact Support.

DEPLOYMENT_DOCUMENT_SIZE_EXCEEDED

You might get this error when the requested deployment document exceeded the maximum size quota. Reduce the size of the deployment document and try the deployment again.

GREENGRASS_ARTIFACT_SIZE_NOT_FOUND

You might get this error when Greengrass can't get the size of a public component artifact. Check the log to see what caused the error, then check the nucleus software update page to see if the issue has been corrected in a later version of the nucleus, or contact Support.

DEPLOYMENT_DOCUMENT_NOT_VALID

You might get this error when the device receives deployment document that isn't valid. Check any additional error codes or check the log to see what caused the error.

EMPTY_DEPLOYMENT_REQUEST

You might get this error when a device receives an empty deployment request. Check the log to see what caused the error, then check the nucleus software update page to see if the issue has been corrected in a later version of the nucleus, or contact Support.

DEPLOYMENT_DOCUMENT_PARSE_ERROR

You might get this error when the deployment request format doesn't match the expected format. Check the log to see what caused the error, then check the nucleus software update page to see if the issue has been corrected in a later version of the nucleus, or contact Support.

COMPONENT_METADATA_NOT_VALID_IN_DEPLOYMENT

You might get this error when the deployment request contains component metadata that isn't valid. Check the log to see what caused the error, then check the nucleus software update page to see if the issue has been corrected in a later version of the nucleus, or contact Support.

Generic errors

These generic errors do not have an associated error type.

DEPLOYMENT_INTERRUPTED

You might get this error when a deployment can't be completed because of a nucleus shutdown or other external event. Check any additional error codes or logs for details.

ARTIFACT_DOWNLOAD_ERROR

You might get this error when there is a problem downloading an artifact. Check any additional error codes or logs for details.

NO_AVAILABLE_COMPONENT_VERSION

You might get this error when a component version doesn't exist in the cloud or locally, or if there is a dependency resolution conflict. Check any additional error codes or logs for details.

COMPONENT_PACKAGE_LOADING_ERROR

You might get this error when an error processing the downloaded artifacts. Check any additional error codes or logs for details.

CLOUD_API_ERROR

You might get this error when an error occurred calling an AWS service API. Check any additional error codes or logs for details.

IO_ERROR

You might get this error when an I/O error occurs during a deployment. Check any additional error codes or logs for details.

COMPONENT_UPDATE_ERROR

You might get this error when a component doesn't update during a deployment. Check any additional error codes or check the log to see what caused the error.

Unknown error

DEPLOYMENT_FAILURE

You might get this error when a deployment fails because an unchecked exception was thrown. Check the log to see what caused the error, then check the nucleus software update page to see if the issue has been corrected in a later version of the nucleus, or contact Support.

DEPLOYMENT_TYPE_NOT_VALID

You might get this error when the deployment type isn't valid. Check the log to see what caused the error, then check the nucleus software update page to see if the issue has been corrected in a later version of the nucleus, or contact Support.

Detailed component status codes

Use the status codes and solutions in these sections to help resolve issues with components when using the Greengrass nucleus version 2.8.0 or later.

Many of the statuses in this topic report additional information in the AWS IoT Greengrass Core logs. These logs are stored on the core device's local file system. There are logs for each individual component. For information on accessing the logs, see Access file system logs.

INSTALL_ERROR

You might get this when an error occurs while running an installation script. The error code is reported in the component log. Check the installation script for errors and deploy your component again.

INSTALL_CONFIG_NOT_VALID

You might get this error when installation of a component couldn't be completed because the `install` section of the recipe isn't valid. Check the installation section of your recipe for errors and try the deployment again.

INSTALL_IO_ERROR

You might get this when an I/O error occurred during installation of a component. Check the component error log for details about the error.

INSTALL_MISSING_DEFAULT_RUNWITH

You might get this error when AWS IoT Greengrass can't determine the user or group to use when installing a component. Check to make sure that the `runWith` section of your installation recipe includes a valid user or group.

INSTALL_TIMEOUT

You might get this error when the installation script did not finish within the configured timeout period. Either increase the `Timeout` period specified in the recipe's `install` section or modify your installation script to finish within the configured timeout.

STARTUP_ERROR

You might get this when an error occurs while running a startup script. The error code is reported in the component log. Check the installation script for errors and deploy your component again.

STARTUP_CONFIG_NOT_VALID

You might get this error when installation of a component couldn't be completed because the `startup` section of the recipe isn't valid. Check the startup section of your recipe for errors and try the deployment again.

STARTUP_IO_ERROR

You might get this when an I/O error occurred during startup of a component. Check the component error log for details about the error.

STARTUP_MISSING_DEFAULT_RUNWITH

You might get this error when AWS IoT Greengrass can't determine the user or group to use when running a component. Check to make sure that the `runWith` section of your startup recipe includes a valid user or group.

STARTUP_TIMEOUT

You might get this error when the startup script did not finish within the configured timeout period. Either increase the `Timeout` period specified in the recipe's `startup` section or modify your startup script to finish within the configured timeout.

RUN_ERROR

You might get this when an error occurs while running a component script. The error code is reported in the component log. Check the run script for errors and deploy your component again.

RUN_MISSING_DEFAULT_RUNWITH

You might get this error when AWS IoT Greengrass can't determine the user or group to use when running a component. Check to make sure that the `runWith` section of your run recipe includes a valid user or group.

RUN_CONFIG_NOT_VALID

You might get this error when a component couldn't be run because the `run` section of the recipe isn't valid. Check the run section of your recipe for errors and try the deployment again.

RUN_IO_ERROR

You might get this when an I/O error occurred while the component is running. Check the component error log for details about the error.

RUN_TIMEOUT

You might get this error when the run script did not finish within the configured timeout period. Either increase the `Timeout` period specified in the recipe's `run` section or modify your run script to finish within the configured timeout.

SHUTDOWN_ERROR

You might get this when an error occurs while shutting down a component script. The error code is reported in the component log. Check the shutdown script for errors and deploy your component again.

SHUTDOWN_TIMEOUT

You might get this error when the shutdown script did not finish within the configured timeout period. Either increase the `Timeout` period specified in the recipe's `shutdown` section or modify your run script to finish within the configured timeout.

UNINSTALL_ERROR

You might get this when an error occurs while running the uninstall script during component removal. The error code is reported in the component log. Check the uninstall script for errors. Uninstall failures do not fail the deployment, and the component is still removed from the device.

UNINSTALL_TIMEOUT

You might get this error when the uninstall script did not finish within the configured timeout period. Either increase the `Timeout` period specified in the recipe's `uninstall` section or modify your uninstall script to finish within the configured timeout. The default timeout is 120 seconds. Uninstall failures do not fail the deployment, and the component is still removed from the device.

Tag your AWS IoT Greengrass Version 2 resources

With tags, you can organize and manage your resources in AWS IoT Greengrass. You can use tags to assign metadata to your resources, and you can use tags in IAM policies to define conditional access to your resources.

> **Note**
>
> Currently, Greengrass resource tags are not supported for AWS IoT billing groups or cost allocation reports.

Using tags in AWS IoT Greengrass V2

You can use tags to categorize your AWS IoT Greengrass resources by purpose, owner, environment, or any other classification for your use case. When you have many resources of the same type, tags help you more readily identify a specific resource.

Each tag consists of a key and an optional value, both of which you define. For example, you could define a set of tags for your core devices that helps you track them by the customers who own the devices. We recommend that you create a set of tag keys that meets your needs for each kind of resource. By using a consistent set of tag keys, you can more easily manage your resources.

Tag with the AWS Management Console

The **Tag Editor** in the AWS Management Console provides a central, unified way for you to create and manage your tags for resources from all AWS services. For more information, see Tag Editor in the *AWS Resource Groups User Guide*.

Tag with the AWS IoT Greengrass V2 API

You can also use the AWS IoT Greengrass V2 API to work with tags. Before you create tags, be aware of tagging restrictions. For more information, see Tag naming and usage conventions in the *AWS General Reference*.

- To add tags when you create a resource, define them in the `tags` property of the resource.
- To add tags to an existing resource, or to update tag values, use the TagResource operation.
- To remove tags from a resource, use the UntagResource operation.

- To retrieve the tags that are associated with a resource, use the ListTagsForResource operation, or describe the resource and inspect its `tags` property.

The following table lists resources that you can tag using the AWS IoT Greengrass V2 API and their corresponding `Create` and `Describe` or `Get` operations.

Taggable AWS IoT Greengrass V2 resources

Resource	Create operation	Describe or get operation
Core device	None. Run the AWS IoT Greengrass Core software on a device to create a core device.	GetCoreDevice
Component	CreateComponentVersion	DescribeComponent, GetComponent
Deployment	CreateDeployment	GetDeployment

Use the following operations to view and manage tags for resources that support tagging:

- TagResource – Adds tags to a resource, or updates an existing tag's value.
- ListTagsForResource – Lists the tags for a resource.
- UntagResource – Removes tags from a resource.

You can add or remove tags for a resource at any time. To change the value of a tag key, add a tag to the resource that defines the same key and the new value. The new value replaces the previous value. You can set a value to an empty string, but you can't set a value to null.

When you delete a resource, tags that are associated with that resource are also deleted.

Using tags with IAM policies

In your IAM policies, you can use resource tags to control user access and permissions. For example, policies can allow users to create only those resources that have a specific tag. Policies can also restrict users from creating or modifying resources that have certain tags.

Note

If you use tags to allow or deny users' access to resources, you should deny users the ability to add or remove those tags for the same resources. Otherwise, a user could circumvent your restrictions and gain access to a resource by modifying its tags.

You can use the following condition context keys and values in the `Condition` element, also called the `Condition` block, of a policy statement.

`greengrassv2:ResourceTag/`*`tag-key`*`: `*`tag-value`*

Allow or deny actions on resources with specific tags.

`aws:RequestTag/`*`tag-key`*`: `*`tag-value`*

Require that a specific tag be used, or not used, when creating or modifying a taggable resource.

`aws:TagKeys: [`*`tag-key`*`, ...]`

Require that a specific set of tag keys be used, or not used, when creating or modifying a taggable resource.

Note

The condition context keys and values in an IAM policy apply only to actions that have a taggable resource as a required parameter. For example, you can set tag-based conditional access for GetComponent.

For more information, see Controlling access to AWS resources using resource tags and IAM JSON policy reference in the *IAM User Guide*.

Creating AWS IoT Greengrass resources with AWS CloudFormation

AWS IoT Greengrass is integrated with AWS CloudFormation, a service that helps you to model and set up your AWS resources so that you can spend less time creating and managing your resources and infrastructure. You create a template that describes all the AWS resources that you want (such as component versions and deployments), and CloudFormation provisions and configures those resources for you.

When you use CloudFormation, you can reuse your template to set up your AWS IoT Greengrass resources consistently and repeatedly. Describe your resources once, and then provision the same resources over and over in multiple AWS accounts and Regions.

AWS IoT Greengrass and CloudFormation templates

To provision and configure resources for AWS IoT Greengrass and related services, you must understand CloudFormation templates. Templates are formatted text files in JSON or YAML. These templates describe the resources that you want to provision in your CloudFormation stacks. If you're unfamiliar with JSON or YAML, you can use CloudFormation Designer to help you get started with CloudFormation templates. For more information, see What is CloudFormation Designer? in the *AWS CloudFormation User Guide*.

AWS IoT Greengrass supports creating component versions and deployments in CloudFormation. For more information, including examples of JSON and YAML templates for component versions and deployments, see the AWS IoT Greengrass resource type reference in the *AWS CloudFormation User Guide*.

ComponentVersion template example

The following is the YAML template for a version of a simple component. The JSON recipe includes line breaks for readability.

```
Parameters:
  ComponentVersion:
    Type: String
Resources:
  TestSimpleComponentVersion:
```

```
    Type: AWS::GreengrassV2::ComponentVersion
    Properties:
      InlineRecipe: !Sub
        - "{\n
            \"RecipeFormatVersion\": \"2020-01-25\",\n
            \"ComponentName\": \"component1\",\n
            \"ComponentVersion\": \"${ComponentVersion}\",\n
            \"ComponentType\": \"aws.greengrass.generic\",\n
            \"ComponentDescription\": \"This\",\n
            \"ComponentPublisher\": \"You\",\n
            \"Manifests\": [\n
            {\n
              \"Platform\": {\n
              \"os\": \"darwin\"\n
            },\n
            \"Lifecycle\": {},\n
            \"Artifacts\": []\n
           },\n
           {\n
             \"Lifecycle\": {},\n
             \"Artifacts\": []\n
           }\n
          ],\n
          \"Lifecycle\": {\n
          \"install\": {\n
            \"script\": \"yuminstallpython\"\n
          }\n
         }\n
        }"
        - { ComponentVersion: !Ref ComponentVersion }
```

Deployment template example

The following is a YAML file defining a simple template for a deployment.

```
Parameters:
  ComponentVersion:
    Type: String
  TargetArn:
    Type: String
Resources:
  TestDeployment:
    Type: AWS::GreengrassV2::Deployment
```

```
    Properties:
      Components:
        component1:
          ComponentVersion: !Ref ComponentVersion
      TargetArn: !Ref TargetArn
      DeploymentName: CloudFormationIntegrationTest
      DeploymentPolicies:
        FailureHandlingPolicy: DO_NOTHING
        ComponentUpdatePolicy:
          TimeoutInSeconds: 5000
          Action: SKIP_NOTIFY_COMPONENTS
        ConfigurationValidationPolicy:
          TimeoutInSeconds: 30000
Outputs:
  TestDeploymentArn:
    Value: !Sub
      - arn:${AWS::Partition}:greengrass:${AWS::Region}:${AWS::AccountId}:deployments:
${DeploymentId}
      - DeploymentId: !GetAtt TestDeployment.DeploymentId
```

Learn more about CloudFormation

To learn more about CloudFormation, see the following resources:

- AWS CloudFormation
- AWS CloudFormation User Guide
- CloudFormation API Reference
- AWS CloudFormation Command Line Interface User Guide

Open source AWS IoT Greengrass Core software

The AWS IoT Greengrass Version 2 edge runtime (nucleus) and other components of the AWS IoT Greengrass Core software are open source. This means that you can review the code to troubleshoot interactions with your applications. You can also customize and extend the AWS IoT Greengrass Core software to meet your specific software and hardware needs.

For information about the open source repositories for the AWS IoT Greengrass Core software, see the aws-greengrass organization on GitHub. Your use of open source software is governed by the open source license in the corresponding GitHub repository.

Your use of the AWS IoT Greengrass Core software and components not subject to an open source license is governed by the AWS Greengrass Core Software License.

Document history for the AWS IoT Greengrass V2 Developer Guide

The following table describes the documentation for this release of AWS IoT Greengrass Version 2.

- **API version:** 2020-11-30

Change	Description	Date
Clarified deployment failure handling for bootstrap deployments	Clarified that the AWS IoT Greengrass Core software ignores the failure handling policy for bootstrap deployments. These deployments always roll back on failure.	July 30, 2026
AWS IoT Greengrass Core v2.18.2 software update	This release provides version 2.18.2 of the Greengrass nucleus component and updates AWS-provided components.	July 20, 2026
AWS IoT Greengrass Core v2.18.1 software update	This release provides version 2.18.1 of the Greengrass nucleus component and updates AWS-provided components.	July 15, 2026
AWS IoT Greengrass Core v2.18.0 software update	This release provides version 2.18.0 of the Greengrass nucleus component and updates AWS-provided components.	July 8, 2026

Greengrass nucleus lite v2.5.1 software update	This release provides version 2.5.1 of the Greengrass nucleus lite component.	May 6, 2026
Greengrass nucleus lite component v2.5.1 released	Version 2.5.1 of the Greengrass nucleus lite component is available. This version includes several bug fixes and improvements.	May 6, 2026
Stream manager v2.3.0 released	Stream manager v2.3.0 is now available. This release updates to the AWS SDK for Java v2 and changes proxy precedence behavior.	April 16, 2026
Greengrass nucleus lite component v2.5.0 released	Version 2.5.0 of the Greengrass nucleus lite component is available. This version adds PKCS#11 support and adds support for using TPM with fleet provisioning. This version also includes several bug fixes.	April 16, 2026
Greengrass CLI v2.17.0 released	Greengrass CLI component v2.17.0 is available.	April 16, 2026
AWS IoT Greengrass Core v2.17.0 software update	This release provides version 2.17.0 of the Greengrass nucleus component and updates AWS-provided components.	April 16, 2026

Greengrass nucleus lite component v2.4.0 released	Version 2.4.0 of the Greengrass nucleus lite component is available. This version adds support for local deployment options and applies artifact permissions from recipes. This version also includes several other bug fixes and improvements.	March 24, 2026
Greengrass nucleus lite component v2.3.3 released	Version 2.3.3 of the Greengrass nucleus lite component is available. This version includes several bug fixes and improvements.	February 26, 2026
Shadow manager v2.3.13 released	Shadow manager v2.3.13 is available. This release fixes an issue where local shadow updates would intermittently fail to sync to the cloud even when sync direction configuration allows it.	February 18, 2026
Greengrass nucleus lite component v2.3.2 released	Version 2.3.2 of the Greengrass nucleus lite component is available. This version updates the version file for correct nucleus version reporting.	February 12, 2026
Greengrass nucleus lite component v2.3.1 released	Version 2.3.1 of the Greengrass nucleus lite component is available. This version includes several bug fixes and improvements.	February 11, 2026

Greengrass CLI v2.16.1 released	Greengrass CLI component v2.16.1 is available.	December 23, 2025
AWS IoT Greengrass Core v2.16.1 software update	This release provides version 2.16.1 of the Greengrass nucleus component and updates AWS-provided components.	December 23, 2025
System log forwarder v2.1.0 released	System log forwarder v2.1.0 is available.	November 6, 2025
Shadow manager v2.3.12 released	Shadow manager v2.3.12 is available.	November 6, 2025
Log manager v2.3.11 released	Log manager component v2.3.11 is available.	November 6, 2025
Greengrass nucleus lite component v2.3.0 released	Version 2.3.0 of the Greengras s nucleus lite component is available. This version adds support for TPM 2.0, a wider range of operating systems, and RestartComponent IPC. This version also includes several other bug fixes and improvements.	November 6, 2025
Greengrass CLI v2.16.0 released	Greengrass CLI component v2.16.0 is available.	November 6, 2025
AWS IoT Greengrass Core v2.16.0 software update	This release provides version 2.16.0 of the Greengrass nucleus component and updates AWS-provided components.	November 6, 2025

AWS IoT Greengrass Core v2.15.1 software update	This release provides version 2.15.1 of the Greengrass nucleus component and updates AWS-provided components.	November 6, 2025
System log forwarder v2.0.1 released	System log forwarder v2.0.1 is available. This version updates the component recipe to properly support aarch64 (arm64) systems.	October 2, 2025
Greengrass nucleus lite component v2.2.2 released	Version 2.2.2 of the Greengrass nucleus lite component is available. This version includes several bug fixes and improvements.	August 18, 2025
System log forwarder v2.0.0 released	System log forwarder component v2.0.0 is available. Initial version released.	July 25, 2025
Greengrass nucleus lite component v2.2.1 released	Version 2.2.1 of the Greengrass nucleus lite component is available. This version fixes an issue where the nucleus fails to obtain TES credentials.	July 25, 2025
Greengrass nucleus lite component v2.2.0 released	Version 2.2.0 of the Greengrass nucleus lite component is available. This version adds support for container image artifact URIs.	July 3, 2025

AWS IoT Greengrass Core v2.15.0 software update	This release provides version 2.15.0 of the Greengrass nucleus component and updates AWS-provided components.	July 3, 2025
Secure tunneling v1.1.3 released	Secure tunneling v1.1.3 is available. This version upgrades the underlying AWS IoT Device Client invoked by the component from version 1.10.0 to version 1.10.1. This version also fixes the GNU C Library (glibc) compatibi lity issues in secure tunneling component version 1.1.2.	May 16, 2025
Greengrass nucleus lite component v2.1.0 released	Version 2.1.0 of the Greengras s nucleus lite component is available. This version adds HTTP proxy support. This version also includes several other bug fixes and improvements.	May 5, 2025
Secret manager v2.2.5 released	Secret manager v2.2.5 is available. This version fixes an issue where a secret doesn't get fetched from the AWS Cloud if it isn't present in the local cache.	April 16, 2025
AWS IoT Greengrass Core v2.14.3 software update	This release provides version 2.14.3 of the Greengrass nucleus component and updates AWS-provided components.	April 11, 2025

Secret manager v2.2.4 released	Secret manager v2.2.4 is available. This version reduces the frequency of writes to the local secret store. Secret manager now writes to the local store only when secrets are updated.	April 8, 2025
Greengrass CLI v2.14.3 released	Greengrass CLI component v2.14.3 is available.	April 8, 2025
Secure tunneling v1.1.2 released	Secure tunneling v1.1.2 is available. This version upgrades the underlying AWS IoT Device Client invoked by the component from version 1.9.0 to version 1.10.0. Secure tunneling v1.1.2 also fixes the payload transfer issue which prevents users from forwardin g large files from Greengrass V2 core devices to the source device through the secure tunnel.	March 27, 2025
Greengrass CLI v2.14.2 released	Greengrass CLI component v2.14.2 is available.	March 27, 2025
AWS IoT Greengrass Core v2.14.2 software update	This release provides version 2.14.2 of the Greengrass nucleus component and updates AWS-provided components.	March 27, 2025

Secret manager v2.2.3 released	Secret manager v2.2.3 is available. This version fixes an issue where secret manager wipes the locally persisted secrets when the core device is offline and the device's security service (such as an HSM) is unavailable.	March 18, 2025
Stream manager v2.2.1 released	Stream manager v2.2.1 is now available. This release fixes an issue where stream manager fails to export messages to Kinesis destinations. This release also improves the performance of stream manager exports to Kinesis destinations.	March 12, 2025
Client device auth component v2.5.3 released	Client device auth component v2.5.3 is available. This release fixes an issue where client devices are unable to connect to the core device due to out of date client certificates.	March 12, 2025
AWS IoT fleet provisioning plugin v1.2.2 released	AWS IoT fleet provisioning plugin v1.2.2 is available. This release adds support for custom core device certificate (`certificatePath`) and private key (`privateKeyPath`) paths.	March 12, 2025

Secure tunneling v1.1.1 released	Secure tunneling v1.1.1 is available. This version adds configuration to support Greengrass nucleus lite.	February 7, 2025
Greengrass CLI v2.14.1 released	Greengrass CLI component v2.14.1 is available.	February 7, 2025
AWS IoT Greengrass Core v2.14.1 software update	This release provides version 2.14.1 of the Greengrass nucleus component and updates AWS-provided components.	February 7, 2025
Greengrass nucleus lite component v2.0.2 released	Version 2.0.2 of the Greengras s nucleus lite component is available. This version includes general bug fixes and improvements.	February 6, 2025
Greengrass nucleus lite component v2.0.1 released	Version 2.0.1 of the Greengras s nucleus lite component is available. This version includes general bug fixes and improvements.	January 28, 2025
Greengrass CLI v2.14.0 released	Greengrass CLI component v2.14.0 is available.	December 24, 2024
Stream manager v2.2.0 released	Stream manager v2.2.0 is now available.	December 16, 2024
Shadow manager v2.3.10 released	Shadow manager v2.3.10 is available.	December 16, 2024

Secure tunneling v1.1.0 released	Secure tunneling v1.1.0 is available. This version adds recipe supports for Greengras s nucleus lite.	December 16, 2024
New nucleus telemetry emitter component	Version 1.0.10 of the nucleus telemetry emitter component is available.	December 16, 2024
Modbus-RTU protocol adapter v2.1.10 released	Modbus-RTU protocol adapter component v2.1.10 is available.	December 16, 2024
Log manager v2.3.9 released	Log manager component v2.3.9 is available.	December 16, 2024
Local debug console v2.4.4 released	Local debug console component v2.4.4 is available . This version includes general bug fixes and improvements.	December 16, 2024
Lambda manager v2.3.5 released	Lambda manager component v2.3.5 is available.	December 16, 2024
IP detector v2.2.1 released	IP detector component v2.2.1 is available.	December 16, 2024
Greengrass nucleus lite component v2.0.0 released	Version 2.0.0 of the Greengras s nucleus lite component is available. This is the initial version.	December 16, 2024
Disk spooler v1.0.5 released	Disk spooler component v1.0.5 is available.	December 16, 2024
Client device auth component v2.5.2 released	Client device auth component v2.5.2 is available.	December 16, 2024

AWS IoT Greengrass Core v2.14.0 software update	This release provides version 2.14.0 of the Greengrass nucleus component, and new AWS IoT Greengrass nucleus lite updates. The AWS IoT Greengrass nucleus lite is a new runtime, available for AWS IoT Greengrass version 2. It provides a reduced memory footprint alternati ve. This is a good option for resource-constrained devices. It implements a subset of the nucleus functionality with increased featured compatibility planned for future releases. The source code is available now on Github. With the nucleus lite runtime you can: • Deploy components to Greengrass core devices. Use the same recipe format, though some advanced features may not be available yet. • Applications deployed as Greengrass components can use the device SDKs to access the supported Greengrass IPC APIs, such as: AWS IoT Core MQTT access, local pub/sub, and Greengrass configuration access. See the compatibi	December 16, 2024

lity chart for the list of supported IPC APIs.

- Some AWS managed components have been updated for nucleus lite support. See the AWS-provided components for a list of existing compatible components.

New features:

- Uses less memory and disk space (less than 5MB of RAM and less than 5MB of storage).
- Components integrate with the host system's service manager (systemd for currently supported Linux platforms).

Things to watch out for:

- AWS IoT Greengrass nucleus lite recipes are case-sensitive. Ensure the correct (keys) casing is used as in the https://docs.aws.amazon.com/greengrass/v2/developerguide/component-recipe-reference.htmlrecipe reference.
- The nucleus lite runtime supports **thing group**

deployments, and does not yet support the (single) **Core device** deployment target type. To deploy to a single Greengrass device, use a thing group with only that one device in it.

- The nucleus lite runtime uses bounded memory resources; functionality which scales according to usage on the classic runtime may fail due to exceeding resources available on lite. This includes a current limitation on max of 50 MQTT subscriptions at a time, and maximum limits on recipe file sizes and deployments. Some of these limits are configurable at compile time if compiling the lite runtime yourself.
- The nucleus lite runtime does not ship with Java. To use components requiring Java, the system will need Java already installed, or a component may be used to install Java.
- We recommend compiling the nucleus lite runtime from source and using your own build tailored for your system. For Yocto

systems, a layer is available to integrate the nucleus lite runtime into your system image.

- Currently the nucleus lite assumes a Linux system using *systemd*, or a container image using *systemd*.
- While you can manage Docker containers with recipe scripts, Greengras s managed container artifacts are not yet available.
- The nucleus lite runtime does not yet have support for keys stored in a PKCS11 module. If your use case requires keys stored on a secure element, the classic runtime can support this use case currently. To prevent leaks of your device credentials, ensure production devices are using full disk encryption.

Alongside the introduction of nucleus lite, we are also releasing nucleus v2.14.0. This update brings significant enhancements to the existing Greengrass nucleus.

	Key features and improvements: • New dual-stack endpoint support enables IPv6 network communication. • Enhanced resilience against nucleus restart failures and directory corruption. • Fixed memory leaks in IPC PubSub subscription closures.	
Stream manager v2.1.13 released	Stream manager v2.1.13 is now available. This release adds support for FIPS endpoint for AWS IoT SiteWise	August 26, 2024
Shadow manager v2.3.9 released	Shadow manager v2.3.9 is available.	August 26, 2024
Modbus-RTU protocol adapter v2.1.9 released	Modbus-RTU protocol adapter component v2.1.9 is available.	August 26, 2024
Log manager v2.3.8 released	Log manager component v2.3.8 is available.	August 26, 2024
Local debug console v2.4.3 released	Local debug console component v2.4.3 is available. This version includes general bug fixes and improvements.	August 26, 2024
Disk spooler v1.0.4 released	Disk spooler component v1.0.4 is available.	August 26, 2024

AWS IoT Greengrass Core v2.13.0 software update	This release provides version 2.13.0 of the Greengrass nucleus component.	August 26, 2024
New nucleus telemetry emitter component	Version 1.0.9 of the nucleus telemetry emitter component is available.	August 23, 2024
Lambda manager v2.3.4 released	Lambda manager component v2.3.4 is available.	August 23, 2024
Greengrass CLI v2.13.0 released	Greengrass CLI component v2.13.0 is available.	August 23, 2024
Client device auth component v2.5.1 released	Client device auth component v2.5.0 is available. This release adds support for FIPS endpoint.	August 23, 2024
Recipe validation	Added a recipe validation feature that will validate a component recipe when creating a component version.	August 15, 2024
IP detector v2.2.0 released	IP detector component v2.2.0 is available. This release adds support for IPv6. You can now use IPv6 for local messaging.	July 29, 2024
Shadow manager v2.3.8 released	Shadow manager v2.3.8 is available. This release fixes an issue where shadow manager creates a deadlock situation during the MQTT client connection.	June 5, 2024
Greengrass CLI v2.12.6 released	Greengrass CLI component v2.12.6 is available.	May 24, 2024

AWS IoT Greengrass Core v2.12.6 software update	This release provides version 2.12.6 of the Greengrass nucleus component and updates AWS-provided components.	May 24, 2024
AWS IoT Device Tester v4.9.4 with GGV2Q v2.5.4 released	Version 4.9.4 of IDT for AWS IoT Greengrass V2 is available. This release includes the AWS IoT Greengrass V2 qualificat ion suite (GGV2Q) v2.5.4, and supports Greengrass nucleus versions 2.12.0, 2.11. 0, 2.10.0, 2.9.5.	May 3, 2024
Secure tunneling v1.0.19 released	Secure tunneling v1.0.19 is available. This version upgrades the underlying AWS IoT Device Client invoked by the component from version 1.8.0 to version 1.9.0. Secure tunneling v1.0.19 increases the concurrent tunnel limit to 20 tunnels on a component level. This new version also increases AWS IoT Greengras s Core IPC timeout from 3 seconds to 10 seconds.	May 1, 2024
Edge connector for Kinesis Video Streams component v1.0.5 released	Version 1.0.5 of the edge connector for Kinesis Video Streams component is available. This version includes general bug fixes and improvements.	April 29, 2024

Greengrass CLI v2.12.5 released	Greengrass CLI component v2.12.5 is available.	April 25, 2024
Client device auth component v2.5.0 released	Client device auth component v2.5.0 is available. This release adds policy variable support for thing names. This release also allows policy resources with wildcards.	April 25, 2024
AWS IoT Greengrass Core v2.12.5 software update	This release provides version 2.12.5 of the Greengrass nucleus component and updates AWS-provided components.	April 25, 2024
AWS IoT Device Tester v4.9.3 with GGV2Q v2.5.3 released	Version 4.9.3 of IDT for AWS IoT Greengrass V2 is available. This release includes the AWS IoT Greengrass V2 qualificat ion suite (GGV2Q) v2.5.3, and supports Greengrass nucleus versions 2.12.0, 2.11. 0, 2.10.0, 2.9.5.	April 5, 2024
Greengrass CLI v2.12.4 released	Greengrass CLI component v2.12.4 is available.	April 2, 2024
AWS IoT Greengrass Core v2.12.4 software update	This release provides version 2.12.4 of the Greengrass nucleus component and updates AWS-provided components.	April 2, 2024

Shadow manager v2.3.7 released	Shadow manager v2.3.7 is available. This release fixes an issue where shadow manager periodically logs a `NullPointerExcepti` on error during a shadow manager sync.	March 27, 2024
Moquette MQTT 3.1.1 broker v2.3.6 released	Moquette MQTT 3.1.1 broker component v2.3.6 is available . This version includes general bug fixes and improvements.	March 27, 2024
Local debug console v2.4.2 released	Local debug console component v2.4.2 is available . This version includes general bug fixes and improvements.	March 27, 2024
Lambda manager v2.3.3 released	Lambda manager component v2.3.3 is available. This version includes general bug fixes and improvements.	March 27, 2024
IP detector v2.1.9 released	IP detector component v2.1.9 is available. This release adjusts the IP acquired step to only send logs at the debug log level.	March 27, 2024

AWS IoT fleet provisioning plugin v1.2.1 released	AWS IoT fleet provisioning plugin v1.2.1 is available. This release fixes an issue where the fleet provision ing plugin is offline during a Greengrass nucleus startup. The fleet provisioning plugin now indefinitely retries MQTT connect calls.	March 27, 2024
AWS IoT Greengrass Core v2.12.3 software update	This release provides version 2.12.3 of the Greengrass nucleus component and updates AWS-provided components.	March 27, 2024
Greengrass CLI v2.12.3 released	Greengrass CLI component v2.12.3 is available.	March 25, 2024
AWS IoT Device Tester v4.9.2 with GGV2Q v2.5.2 released	Version 4.9.2 of IDT for AWS IoT Greengrass V2 is available. This release includes the AWS IoT Greengrass V2 qualificat ion suite (GGV2Q) v2.5.2, and supports Greengrass nucleus versions 2.12.0, 2.11. 0, 2.10.0, 2.9.5.	March 18, 2024
AWS IoT Greengrass Core v2.12.2 software update	This release provides version 2.12.2 of the Greengrass nucleus component and updates AWS-provided components.	February 15, 2024

Shadow manager v2.3.6 released	Shadow manager v2.3.6 is available. This release fixes an issue where shadow properties that are deleted through AWS Cloud updates while the device is offline continue to exist in the local shadow after regaining connectivity.	February 14, 2024
Lambda launcher v2.0.13 released	Version 2.0.13 of the Lambda launcher component is available. This release includes general bug fixes and improvements.	February 14, 2024
Disk spooler v1.0.3 released	Disk spooler component v1.0.3 is available. This release improves performance by reusing database connections.	February 14, 2024
Greengrass Development Kit CLI v1.6.2	Version 1.6.2 of the Greengrass Development Kit CLI is available. This version fixes an issue where Windows gradlew.bat does not work due to the relative path. This version also contains additional improvements.	January 16, 2024

New CloudTrail data events	You can now log AWS CloudTrail data events to get information about resource operations such as getting a component or the configuration of a deployment. Use these events to gain insight into the operation of your Greengrass devices.	December 20, 2023
Stream manager v2.1.12 released	Stream manager v2.1.12 is now available. This release changes the order that Greengrass uses to select a set of credentials for AWS service calls.	December 8, 2023
MQTT bridge v2.3.1 released	MQTT bridge v2.3.1 is available. This release fixes a rare issue where the local MQTT client gets into a disconnect loop.	December 8, 2023
Disk spooler v1.0.2 released	Disk spooler component v1.0.2 is available. This release fixes an issue where the MQTT message format field isn't persisted in certain cases.	December 8, 2023

Client device auth component v2.4.5 released	Client device auth component v2.4.5 is available. This release adds support for wildcards at the end of thing names in a selection rule and fixes an issue where certifica tes are not updated with new connectivity info in certain cases.	December 8, 2023
AWS IoT Greengrass Core v2.12.1 software update	This release provides version 2.12.1 of the Greengrass nucleus component and updates AWS-provided components.	December 8, 2023
Greengrass Development Kit CLI v1.6.1	Version 1.6.1 of the Greengras s Development Kit CLI is available. This version contains bug fixes and improvements.	December 6, 2023
Recipe validation	Added a recipe validation feature that will validate a component recipe when creating a component version.	November 16, 2023
Publisher-supported components	AWS IoT Greengrass now offers Publisher-supporte d components. These components are developed, offered, and serviced by third-party vendors.	November 16, 2023
Greengrass Testing Framework v1.2.0 released	Greengrass Testing Framework v1.2.0 is available.	November 15, 2023

Greengrass Development Kit CLI v1.6.0	Version 1.6.0 of the Greengrass Development Kit CLI is available. This version adds a recipe validation check against the Greengrass recipe schema during the `component build` and `component publish` commands. This update helps developers to identify actionable issues within their component recipes earlier in the component creation process. This version also adds a confidence test suite to the template that can be pulled down by the `test-e2e init` command. This confidence test suite includes eight generic tests that can be used and extended to fit basic component testing needs.	November 15, 2023
AWS IoT Device Tester v4.9.1 supports Greengrass nucleus version 2.12.0	Version 4.9.1 of IDT for AWS IoT Greengrass V2 now supports Greengrass nucleus version 2.12.0.	November 7, 2023
AWS IoT Greengrass Core v2.12.0 software update	This release provides version 2.12.0 of the Greengrass nucleus component and updates AWS-provided components.	November 7, 2023

Operate a Greengrass core device in VPC	Operating a Greengrass core device in VPC is available . This feature enables you to perform deployments in VPC without public internet access.	November 3, 2023
Greengrass CLI v2.12.0 released	Greengrass CLI component v2.12.0 is available.	October 30, 2023
Stream manager v2.1.10 released	Stream manager v2.1.10 is now available. This release fixes an issue where the HTTPS proxy configuration doesn't trust the Greengrass CA certificate chain.	October 26, 2023
Lambda launcher v2.0.12 released	Version 2.0.12 of the Lambda launcher component is available. This release fixes an issue where the Lambda launcher could throw an error if the previous process was not stopped properly.	October 26, 2023

Greengrass Development Kit CLI v1.5.0	Version 1.5.0 of the Greengrass Developme nt Kit CLI is available. This version updates the patterns recognized by the `excludes` build option when `build_system` is `zip`. This version will now recognize glob patterns which match pathnames based on their wildcard characters. This enables custom specifica tion of which directories to exclude from.	October 26, 2023
Shadow manager v2.3.4 released	Shadow manager v2.3.4 is available. This release adds support for null and empty shadow state documents.	October 18, 2023
Log manager v2.3.6 released	Log manager component v2.3.6 is available.	October 18, 2023
Local debug console v2.4.0 released	Local debug console component v2.4.0 is available.	October 18, 2023
Lambda manager v2.3.1 released	Lambda manager component v2.3.1 is available.	October 18, 2023
Greengrass CLI v2.11.3 released	Greengrass CLI component v2.11.3 is available.	October 18, 2023
AWS IoT Greengrass Core v2.11.3 software update	This release provides version 2.11.3 of the Greengrass nucleus component and updates AWS-provided components.	October 18, 2023

Secure tunneling v1.0.17 released	Secure tunneling v1.0.17 is available.	October 4, 2023
Greengrass Development Kit CLI v1.4.0	Version 1.4.0 of the Greengrass Development Kit CLI is available. This version adds a new `config` command that starts an interactive prompt to modify fields within an existing GDK configuration file. This version also modifies the `gdk component build` and `gdk component publish` commands to verify that the recipe size is within Greengrass requirements (<=16000 bytes) before proceeding.	October 2, 2023
Moquette MQTT 3.1.1 broker v2.3.5 released	Moquette MQTT 3.1.1 broker component v2.3.5 is available. This version updates Moquette to version 0.17.	September 28, 2023
MQTT bridge v2.3.0 released	MQTT bridge v2.3.0 is available. This release adds MQTT 5 support for bridging between AWS IoT Core and local MQTT sources.	September 28, 2023
Lambda manager v2.3.0 released	Lambda manager component v2.3.0 is available.	September 15, 2023

Lambda launcher v2.0.11 released	Version 2.0.11 of the Lambda launcher component is available. This version supports Lambda Manager 2.3.0.	September 15, 2023
Moquette MQTT 3.1.1 broker v2.3.4 released	Moquette MQTT 3.1.1 broker component v2.3.4 is available.	September 1, 2023
Greengrass Testing Framework	GTF is a collection of building blocks to support end-to-end automation. It enables AWS IoT Greengrass Version 2 internal customers to use the same testing framework that the service team uses for qualifying software changes, automated acceptance, and quality assurance purposes.	August 11, 2023
AWS IoT Greengrass Core v2.11.2 software update	This release provides version 2.11.2 of the Greengrass nucleus component and updates AWS-provided components.	August 9, 2023
Greengrass Development Kit CLI v1.3.0	Version 1.3.0 of the Greengras s Development Kit CLI is available. This version adds a new `test-e2e` command to support end-to-end testing of components using Open Test Framework.	July 21, 2023

AWS IoT Greengrass Core v2.11.1 software update	This release provides version 2.11.1 of the Greengrass nucleus component and updates AWS-provided components.	July 21, 2023
Disk spooler v1.0.0 released	Disk spooler component v1.0.0 is available.	June 28, 2023
AWS IoT Greengrass Core v2.11.0 software update	This release provides version 2.11.0 of the Greengrass nucleus component and updates AWS-provided components.	June 28, 2023
AWS IoT Greengrass Core v2.10.3 software update	This release provides version 2.10.3 of the Greengrass nucleus component and updates AWS-provided components.	June 21, 2023
AWS IoT Greengrass Core v2.10.2 software update	This release provides version 2.10.2 of the Greengrass nucleus component and updates AWS-provided components.	June 5, 2023
AWS IoT Greengrass Core v2.10.1 software update	This release provides version 2.10.1 of the Greengrass nucleus component and updates AWS-provided components.	May 11, 2023
AWS IoT Greengrass Core v2.10.0 software update	This release provides version 2.10.0 of the Greengrass nucleus component and updates AWS-provided components.	May 9, 2023

SageMaker AI Edge Manager discontinued	Amazon SageMaker AI Edge Manager component is being discontinued on April 26, 2024.	April 28, 2023
AWS IoT Greengrass Core v2.9.6 software update	This release provides version 2.9.6 of the Greengrass nucleus component and updates AWS-provided components.	April 20, 2023
Log manager v2.3.2 released	Log manager component v2.3.2 is available.	April 19, 2023
Stream manager v2.1.4 released	Stream manager v2.1.4 is now available. This release fixes an issue where entries for the same property asset with the same timestamp within a single batch return `ConflictingOperationException` from the SiteWise API which causes stream manager to continuously retry. This release also updates the default connection timeout from 3 seconds to 1 minute.	April 13, 2023
Greengrass Development Kit CLI v1.2.3	Version 1.2.3 of the Greengrass Development Kit CLI is available. This version contains bug fixes.	April 13, 2023

Client device auth component v2.4.0 released	Client device auth component v2.4.0 is available. This release adds support for client device auth to emit operational metrics that can be displayed on the Greengrass Client Device dashboard.	April 10, 2023
Greengrass Development Kit CLI v1.2.2	Version 1.2.2 of the Greengrass Development Kit CLI is available. This version contains improvements and bug fixes.	April 7, 2023
AWS IoT Greengrass Core v2.9.5 software update	This release provides version 2.9.5 of the Greengrass nucleus component and updates AWS-provided components.	March 30, 2023
Stream manager v2.1.3 released	Stream manager v2.1.3 is now available. This release fixes a startup issue on Windows OS when running as the SYSTEM user.	March 7, 2023
Modbus-RTU protocol adapter v2.1.5 released	Modbus-RTU protocol adapter component v2.1.5 is available. This release fixes an issue with the `ReadDiscreteInput` operation.	March 7, 2023

Client device auth component v2.3.2 released	Client device auth component v2.3.2 is available. This release adds support for caching hostname informati on so that the component correctly generates certificate subjects when restarted while offline.	March 7, 2023
AWS IoT Device Tester v4.7.0 supports Greengrass nucleus version 2.9.4	Version 4.7.0 of IDT for AWS IoT Greengrass V2 now supports Greengrass nucleus version 2.9.4.	March 2, 2023
Greengrass Command Line Interface v1.2.0 released	Greengrass Command Line Interface v1.2.0 is available.	February 28, 2023
AWS IoT Greengrass Core v2.9.4 software update	This release provides version 2.9.4 of the Greengrass nucleus component and updates AWS-provided components.	February 24, 2023
Shadow manager v2.3.1 released	Shadow manager v2.3.1 is available. This release fixes a condition that may prevent cloud shadow updates from syncing. This release also fixes an issue where changes to named shadow sync configura tion applies to only one named shadow.	February 21, 2023
AWS IoT Device Tester v4.7.0 supports Greengrass nucleus version 2.9.3	Version 4.7.0 of IDT for AWS IoT Greengrass V2 now supports Greengrass nucleus version 2.9.3.	February 9, 2023

IAM best practices updated	Updated guide to align with the IAM best practices . For more information, see Security best practices in IAM.	February 3, 2023
AWS IoT Greengrass Core v2.9.3 software update	This release provides version 2.9.3 of the Greengrass nucleus component and updates AWS-provided components.	February 1, 2023
Log manager v2.3.1 released	Log manager v2.3.1 is available.	January 27, 2023
AWS IoT Device Tester v4.7.0 supports Greengrass nucleus version 2.9.2	Version 4.7.0 of IDT for AWS IoT Greengrass V2 now supports Greengrass nucleus version 2.9.2.	January 3, 2023
Shadow manager v2.3.0 released	Shadow manager v2.3.0 is available. This release fixes an issue that might prevent shadows from syncing when a device stores the Greengras s device private key in a hardware security module.	December 29, 2022
AWS IoT fleet provisioning plugin v1.2.0 released	AWS IoT fleet provisioning plugin v1.2.0 is available. This release adds support for device provisioning via certificate signing request with configurable private key path.	December 22, 2022

AWS IoT Greengrass Core v2.9.2 software update	This release provides version 2.9.2 of the Greengrass nucleus component and updates AWS-provided components.	December 22, 2022
AWS IoT Device Tester v4.7.0 with GGV2Q v2.5.0 released	Version 4.7.0 of IDT for AWS IoT Greengrass V2 is available. This release includes the AWS IoT Greengrass V2 qualificat ion suite (GGV2Q) v2.5.0, and supports Greengrass nucleus versions 2.9.1, 2.9.0, 2.8.1, 2.8.0, 2.7.0, and 2.6.0.	December 13, 2022
Shadow manager v2.2.4 released	Fixes an issue where the validation of the shadow's size wasn't consistent with the cloud when updating the local shadow document. This also fixes an issue where the shadow manager stops listening to configuration updates if a deployment performs a RESET on the configuration nodes.	December 8, 2022
Log manager v2.3.0 released	Log manager component v2.3.0 is available.	November 18, 2022
AWS IoT Device Tester v4.5.11 supports Greengrass nucleus version 2.9.1	Version 4.5.11 of IDT for AWS IoT Greengrass V2 now supports Greengrass nucleus version 2.9.1.	November 18, 2022

AWS IoT Greengrass Core v2.9.1 software update	This release provides version 2.9.1 of the Greengrass nucleus component and updates AWS-provided components.	November 18, 2022
AWS IoT Device Tester v4.5.11 supports Greengrass nucleus version 2.9.0	Version 4.5.11 of IDT for AWS IoT Greengrass V2 now supports Greengrass nucleus version 2.9.0.	November 17, 2022
Stream manager v2.1.2 released	Stream manager v2.1.2 is now available. This release fixes an issue on Windows OS that use a non-English language.	November 15, 2022
AWS IoT Greengrass Core v2.9.0 software update	This release provides version 2.9.0 of the Greengrass nucleus component and updates AWS-provided components.	November 15, 2022
AWS IoT Device Tester v4.5.11 supports Greengrass nucleus version 2.8.1	Version 4.5.11 of IDT for AWS IoT Greengrass V2 now supports Greengrass nucleus version 2.8.1.	October 19, 2022
AWS IoT Device Tester v4.5.11 with GGV2Q v2.4.1 released	Version 4.5.11 of IDT for AWS IoT Greengrass V2 is available. This release includes the AWS IoT Greengrass V2 qualificat ion suite (GGV2Q) v2.4.1, and supports Greengrass nucleus versions 2.8.0, 2.7.0, and 2.6.0.	October 13, 2022

AWS IoT Greengrass Core v2.8.1 software update	This release provides version 2.8.1 of the Greengrass nucleus component and updates AWS-provided components.	October 13, 2022
AWS IoT Greengrass Core v2.8.0 software update	This release provides version 2.8.0 of the Greengrass nucleus component and updates AWS-provided components.	October 7, 2022
Added CloudFormation support for deployments	CloudFormation now supports AWS IoT Greengrass deployments as a resource.	October 6, 2022
SageMaker AI Edge Manager v1.3.0 released	Amazon SageMaker AI Edge Manager component v1.3.0 is available. This release adds support for this component to set the disk size for the TensorRT model cache, and improves prediction concurrency to make better use of device accelerator engines such as GPUs.	September 1, 2022
Use interprocess communication (IPC) client V2	Added information about IPC client V2, which reduces the amount of code that you need to write to use IPC operations and helps avoid common errors that can occur with IPC client V1.	August 12, 2022

AWS IoT Device Tester v4.5.8 with GGV2Q v2.4.0 released	Version 4.5.8 of IDT for AWS IoT Greengrass V2 is available. This release includes the AWS IoT Greengrass V2 qualification suite (GGV2Q) v2.4.0, and supports Greengrass nucleus versions 2.7.0, 2.6.0, and 2.5.6.	August 12, 2022
SageMaker AI Edge Manager v1.2.0 released	Amazon SageMaker AI Edge Manager component v1.2.0 is available. This release adds support for this component to automatically retrieve SageMaker AI Neo-compiled models that you upload to Amazon S3, so you can deploy new models without needing to create a AWS IoT Greengrass deployment.	August 3, 2022
AWS IoT Device Tester v4.5.3 supports Greengrass nucleus version 2.7.0	Version 4.5.3 of IDT for AWS IoT Greengrass V2 now supports Greengrass nucleus version 2.7.0.	August 1, 2022
Stream manager v2.1.0 released	Stream manager v2.1.0 is now available. This release includes support for you to send telemetry metrics to Amazon EventBridge.	July 28, 2022

AWS IoT Greengrass Core v2.7.0 software update	This release provides version 2.7.0 of the Greengrass nucleus component and updates AWS-provided components. It includes support for you to send telemetry metrics to Amazon EventBridge.	July 28, 2022
IoT SiteWise publisher v2.2.0 released	IoT SiteWise publisher component v2.2.0 is available . This release updates the component to compress data before sending it to the AWS IoT SiteWise service, which reduces bandwidth usage by up to 75 percent.	July 19, 2022
Tutorial: Develop a component that interacts with client device shadows	Added a new module to Tutorial: Interact with local IoT devices over MQTT that you can follow to learn how to develop a component that interacts with client device shadows.	July 18, 2022
Choose a local MQTT broker	Added information about how to choose a local MQTT broker where client devices connect to a core device.	July 18, 2022
AWS IoT Device Tester v4.5.3 supports Greengrass nucleus version 2.6.0	Version 4.5.3 of IDT for AWS IoT Greengrass V2 now supports Greengrass nucleus version 2.6.0.	June 29, 2022

AWS IoT Greengrass Core v2.6.0 software update	This release provides version 2.6.0 of the Greengrass nucleus component and updates AWS-provided components. It includes support for client device shadows and a local MQTT 5 broker for client devices. It also includes support for wildcards in local publish/subscribe topics, recipe variables in component configurations, and wildcards in IPC authorization policies. These features enable you to more easily develop and configure component s that you deploy to fleets of core devices. This release also includes support for components to use IPC operations that manage local deployments and components on a core device.	June 27, 2022
Client device component updates	Client device auth v2.1.0, MQTT broker (Moquette) v2.1.0, MQTT bridge v2.1.1, and IP detector v2.1.2 are available. This release improves certificate rotation, improves MQTT broker performance, and fixes issues with how these component s handle configuration reset updates.	June 14, 2022

AWS IoT Device Tester v4.5.3 supports Greengrass nucleus version 2.5.6	Version 4.5.3 of IDT for AWS IoT Greengrass V2 now supports Greengrass nucleus version 2.5.6.	June 1, 2022
AWS IoT Greengrass Core v2.5.6 software update	This release provides version 2.5.6 of the Greengrass nucleus component and updates AWS-provided components. It includes support for hardware security modules with ECC keys. It also includes other bug fixes and improvements.	May 31, 2022
AWS IoT fleet provisioning plugin v1.1.0 released	AWS IoT fleet provisioning plugin v1.1.0 is available. This release adds support for additional file path formats when you configure the plugin on Windows devices.	May 12, 2022
New Lambda runtimes released	Added support for new Lambda runtimes: Python 3.9, Java 11, and NodeJS 14.	May 10, 2022
Develop a Greengrass component that defers component updates	Added a tutorial that you can follow to learn how to develop a Greengras s component that defers component updates from deployments. You might want to delay an update when a device has a low battery level or while it runs a process that can't be interrupted, for example.	May 4, 2022

CloudWatch metrics v3.1.0 and AWS IoT Device Defender v3.1.0 released	CloudWatch metrics component v3.1.0 and AWS IoT Device Defender component v3.1.0 are available. These releases add support for HTTPS network proxy configura tions. For more information, see Connect on port 443 or through a network proxy and Enable the core device to trust an HTTPS proxy.	April 27, 2022
Migrate from AWS IoT Greengrass Version 1	Added a guide that you can follow to to migrate from AWS IoT Greengrass V1 to AWS IoT Greengrass V2.	April 26, 2022
AWS IoT Device Tester v4.5.3 with GGV2Q v2.3.1 updated and IDT v4.5.1 with GGV2Q v2.3.0 added to supported versions	Version 4.5.3 of IDT for AWS IoT Greengrass V2 with AWS IoT Greengrass V2 qualifica tion suite (GGV2Q) v2.3.1 has been updated to include support for Greengrass nucleus versions 2.5.5, 2.5.4, and 2.5.3. This update also includes IDT 4.5.1 with AWS IoT Greengrass V2 qualifica tion suite (GGV2Q) v2.3.0 as a supported version. IDT 4.5.1 with AWS IoT Greengrass V2 qualification suite (GGV2Q) v2.3.0 supports Greengrass nucleus version 2.5.3.	April 25, 2022

Modbus-RTU protocol adapter v2.1.0 released	Modbus-RTU protocol adapter component v2.1.0 is available . This release adds new parameters that you can specify to configure serial communication with Modbus RTU devices.	April 20, 2022
CloudWatch metrics v2.1.0, Firehose v2.1.0, and Amazon SNS v2.1.0 released	CloudWatch metrics component v2.1.0, Firehose component v2.1.0, and Amazon SNS component v2.1.0 are available. These releases add support for HTTPS network proxy configurations. For more information, see Connect on port 443 or through a network proxy and Enable the core device to trust an HTTPS proxy.	April 19, 2022
AWS IoT Device Tester v4.5.3 with GGV2Q v2.3.1 released	Version 4.5.3 of IDT for AWS IoT Greengrass V2 is available. This release includes the AWS IoT Greengrass V2 qualificat ion suite (GGV2Q) v2.3.1, and supports Greengrass nucleus version 2.5.5.	April 15, 2022

AWS IoT Greengrass Core v2.5.5 software update	This release provides version 2.5.5 of the Greengrass nucleus component and updates AWS-provided components. It adds support for Windows devices that use a display language other than English. It also fixes an issue where the core device didn't report its status to the AWS IoT Greengrass cloud service after provisioning in certain scenarios.	April 6, 2022
AWS IoT Greengrass Core v2.5.4 software update	This release provides version 2.5.4 of the Greengrass nucleus component and updates AWS-provided components. It includes bug fixes and improvements.	March 23, 2022
Download AWS IoT Device Tester programmatically	Added information about how to download IDT for AWS IoT Greengrass V2 programmatically.	March 15, 2022
Greengrass Development Kit CLI v1.1.0	Version 1.1.0 of the Greengrass Development Kit CLI is available. This version adds new arguments to the `component init` and `component publish` commands. This version also updates the `component publish` command to build the component if it isn't built.	February 24, 2022

Shadow manager v2.1.0 released	Shadow manager component v2.1.0 is available. This release adds the option to configure the interval where the component syncs shadows with AWS IoT Core. For example, you can specify a longer interval to reduce bandwidth usage and charges.	February 3, 2022
Dockerfile and Docker images for AWS IoT Greengrass Core software v2.5.3	The Dockerfile and Docker image for AWS IoT Greengrass Core software v2.5.3 are now available.	January 12, 2022
AWS IoT Device Tester v4.5.1 with GGV2Q v2.3.0 released	Version 4.5.1 of IDT for AWS IoT Greengrass V2 is available . This release includes the AWS IoT Greengrass V2 q ualification suite (GGV2Q) v2.3.0, and supports validatin g and qualifying Linux-based devices that use a hardware security module (HSM) to store the private key and certificate used by AWS IoT Greengrass Core software.	January 11, 2022

AWS IoT Greengrass Core v2.5.3 software update	This release provides version 2.5.3 of the Greengrass nucleus component and updates AWS-provided components. It includes support for you to configure the AWS IoT Greengrass Core software to use a private key and certificate that you securely store in a hardware security module (HSM).	January 6, 2022
Dockerfile and Docker images for AWS IoT Greengrass Core software v2.5.2	The Dockerfile and Docker image for AWS IoT Greengrass Core software v2.5.2 are now available.	December 20, 2021
AWS IoT Device Tester v4.4.1 with GGV2Q v2.2.1 released	Version 4.4.1 of IDT for AWS IoT Greengrass V2 is available. This release includes the AWS IoT Greengrass V2 qualificat ion suite (GGV2Q) v2.2.1, and supports Greengrass nucleus version 2.5.2 for device qualification.	December 12, 2021
Perform machine learning inference using Amazon Lookout for Vision	Added information about how to perform machine learning inference using Lookout for Vision on Greengrass core devices. Lookout for Vision uses computer vision to find visual defects in industrial products.	December 8, 2021

AWS IoT Device Tester v4.4.1 with GGV2Q v2.2.0 released	Version 4.4.1 of IDT for AWS IoT Greengrass V2 is available. This release includes the AWS IoT Greengrass V2 qualification suite (GGV2Q) v2.2.0, and supports Greengrass nucleus version 2.5.2 for device qualification.	December 6, 2021
AWS IoT Greengrass Core v2.5.2 software update	This release provides version 2.5.2 of the Greengrass nucleus component and updates AWS-provided components. It fixes an issue with the Windows service that occurs after the Greengrass nucleus updates. It also includes support for the AWS IoT Device Defender component on Windows devices.	December 3, 2021
New edge connector for Kinesis Video Streams component	Version 1.0.0 of the edge connector for Kinesis Video Streams component is available. This AWS-provided reads video feeds from local cameras and publishes the streams to Kinesis Video Streams. This component integrates with AWS IoT TwinMaker, which enables you to view and manage video streams and other data in Grafana dashboards.	November 30, 2021

Manage Greengrass core devices with AWS Systems Manager	Added information about how to manage Greengrass core devices with AWS Systems Manager. Systems Manager is an AWS service that enables you to view operational data, automate operation tasks, and maintain security and compliance.	November 29, 2021
Greengrass Development Kit CLI	Added information about the AWS IoT Greengrass Development Kit Command-Line Interface (GDK CLI), which is a tool that you can download to your local development computer to help you develop custom Greengrass components. You can use the GDK CLI to create, build, and publish custom components.	November 29, 2021
Community-provided Greengrass components	Added information about the Greengrass Software Catalog, which is an index of Greengrass component s that are developed by the Greengrass community. From this catalog, you can download, modify, and deploy components to create your Greengrass applications.	November 29, 2021

AWS IoT Greengrass Core v2.5.1 software update	This release provides version 2.5.1 of the Greengrass nucleus component and updates AWS-provided components. It includes support for 32-bit Java on Windows devices. It also fixes issues with the new thing group removal behavior and loading system environment variables on Windows devices.	November 23, 2021
AWS IoT Device Tester v4.4.0 with GGV2Q v2.1.0 released	Version 4.4.0 of IDT for AWS IoT Greengrass V2 is available. This release includes the AWS IoT Greengrass V2 qualificat ion suite (GGV2Q) v2.1.0, and supports qualification of Windows-based Greengrass devices running Greengrass nucleus version 2.5.0.	November 19, 2021
AWS IoT Greengrass Core v2.5.0 software update	This release provides version 2.5.0 of the Greengrass nucleus component and updates AWS-provided components. It includes support for running the AWS IoT Greengrass Core software on Windows devices. It also changes the thing group removal behavior and adds support for HTTPS proxies.	November 12, 2021

SageMaker AI Edge Manager v1.1.0 released	Amazon SageMaker AI Edge Manager component v1.1.0 is available. This release adds support for Greengrass core devices running Amazon Linux 2, and adds a new configuration parameter to specify the location of the capture data folder on your device.	November 3, 2021
Cross-service confused deputy prevention update	AWS IoT Greengrass V2 supports using the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in IAM resource policies to prevent the confused deputy problem.	November 1, 2021
Client device component updates	Client device auth v2.0.3, IP detector v2.1.0, MQTT bridge v2.1.0, and MQTT broker (Moquette) v2.0.2 are available. This release adds full support for non-default MQTT broker ports and includes other bug fixes and improvements.	October 28, 2021

Shadow manager v2.0.4 released	Shadow manager component v2.0.4 is available. This release fixes an issue that caused shadow manager to delete newly created versions of any shadow that was previously deleted. Beginning with this release, the `DeleteThingShadow` IPC operation increments the shadow version.	October 20, 2021
Log manager v2.2.0 released	Log manager component v2.2.0 is available. Log manager now supports using a configuration map to provide component log configurations.	October 20, 2021
Lambda manager v2.1.4 released	Lambda manager component v2.1.4 is available. This release fixes an issue that caused Lambda functions that use NodeJS runtimes to process only one message.	October 20, 2021
Use interprocess communication, AWS credentials, and stream manager in Docker container components	Added information about how to use interprocess communication (IPC), AWS credentials, and stream manager in your custom Docker container components.	October 19, 2021

New nucleus telemetry emitter component	Version 1.0.0 of the nucleus telemetry emitter component is available. This AWS-provided component gathers system health telemetry data and publishes it continually to a local topic and an AWS IoT Core MQTT topic.	September 30, 2021
Allow device traffic through a proxy or firewall	Added information about the endpoints and ports that Greengrass core device use, so you can restrict traffic as a security measure.	September 16, 2021
AWS IoT Device Tester v4.2.0 with GGV2Q v2.0.1 released	Version 4.2.0 of IDT for AWS IoT Greengrass V2 has been updated with AWS IoT Greengrass V2 qualification suite (GGV2Q) v2.0.1. This release supports Greengrass nucleus version 2.4.0 for device qualification.	August 31, 2021
Updated machine learning installer components	DLR installer component v1.6.5 and TensorFlow Lite installer component v2.5.4 are available. These component versions include the new `UseInstaller` configuration parameter that lets you disable the default installation script.	August 30, 2021

Embedded Linux support for AWS IoT Greengrass	The BitBake recipe for AWS IoT Greengrass V2 is available in the `meta-aws` project on GitHub. You can use this recipe to build a custom Linux-based operating system using the Yocto Project.	August 20, 2021
Code integrity	Added information about how AWS IoT Greengrass V2 verifies the integrity of software that Greengrass core devices download from the AWS Cloud.	August 19, 2021
VPC endpoints (AWS PrivateLink)	AWS IoT Greengrass now supports interface VPC endpoints (AWS PrivateLink) for the AWS IoT Greengrass control plane. You can establish a private connection between your VPC and the AWS IoT Greengrass control plane.	August 16, 2021
Stream manager v2.0.12 released	Stream manager v2.0.12 is now available. This release fixes an issue that prevented upgrades from version 2.0.7 of the stream manager component to a version between v2.0.8 and v2.0.11.	August 10, 2021
Dockerfile and Docker images for AWS IoT Greengrass Core software v2.4.0	The Dockerfile and Docker image for AWS IoT Greengrass Core software v2.4.0 are now available.	August 9, 2021

AWS IoT Greengrass Core v2.4.0 software update	This release provides version 2.4.0 of the Greengrass nucleus component and updates AWS-provided components. It includes support for component system resource limits, IPC operations to pause and resume components, and provisioning plugins.	August 3, 2021
New AWS IoT SiteWise components	Added the following AWS-provided components for AWS IoT SiteWise: IoT SiteWise OPC-UA collector, IoT SiteWise publisher, and IoT SiteWise processor.	July 29, 2021
AWS IoT Device Tester v4.2.0 with GGV2Q v2.0.0 released	Version 4.2.0 of IDT for AWS IoT Greengrass V2 is available . This release includes the AWS IoT Greengrass V2 q ualification suite (GGV2Q) v2.0.0 and includes support for optional qualification tests for Docker components, machine learning, and stream manager.	July 14, 2021
AWS IoT Greengrass Core IPC library available in AWS IoT Device SDK for C++ v2	Version 1.13.0 of the AWS IoT Device SDK for C++ v2 supports AWS IoT Greengrass Core IPC, so you can develop components in C++ that interact with the AWS IoT Greengrass Core software.	July 14, 2021

SageMaker AI Edge Manager component v1.0.2 released	Amazon SageMaker AI Edge Manager component v1.0.2 is available. This release updates the installation script in the component lifecycle. Your core devices must now have Python 3.6 or later, including `pip` for your version of Python, installed on the device before you deploy this component.	July 12, 2021
Support update for AWS IoT Device Tester for AWS IoT Greengrass V2	IDT for AWS IoT Greengrass V2 version 4.1.0 now supports using Greengrass nucleus version 2.3.0 for device qualification.	July 8, 2021
Dockerfile and Docker images for AWS IoT Greengrass Core software v2.3.0	The Dockerfile and Docker image for AWS IoT Greengrass Core software v2.3.0 are now available.	July 7, 2021
AWS managed policies	Added information about AWS managed policies for AWS IoT Greengrass.	July 2, 2021
New recommended JVM options	Added information about recommended JVM options to control memory allocation for AWS IoT Greengrass Core software.	June 30, 2021

AWS IoT Greengrass Core v2.3.0 software update	This release provides version 2.3.0 of the Greengrass nucleus component and updates AWS-provided components. It includes support for large component configuration documents in deployments.	June 29, 2021
Dockerfile and Docker images for AWS IoT Greengrass Core software v2.2.0	The Dockerfile and Docker image for AWS IoT Greengrass Core software v2.2.0 are now available.	June 28, 2021
AWS IoT Device Tester v4.1.0 with GGV2Q v1.1.1 released	Version 4.1.0 of IDT for AWS IoT Greengrass V2 is available. This release includes the AWS IoT Greengrass V2 qualificat ion suite (GGV2Q) v1.1.1 and supports using Greengrass nucleus v2.2.0, v2.1.0, and v2.0.5 for device qualification.	June 18, 2021
AWS IoT Greengrass Core v2.2.0 software update	This release provides version 2.2.0 of the Greengrass nucleus component and updates AWS-provided components. It includes components that you can deploy to add support for client devices and add the local shadow service.	June 18, 2021

Lambda launcher v2.0.6 released	Version 2.0.6 of the Lambda launcher component is available. This version includes performance improvements and bug fixes.	June 13, 2021
New SageMaker AI Edge Manager component released	Version 1.0.0 of the Amazon SageMaker AI Edge Manager component is available for AWS IoT Greengrass. This component installs the SageMaker AI Edge Manager agent binary on Greengrass core devices.	June 10, 2021
Component types	Added information about component types in AWS IoT Greengrass. The component type specifies how the AWS IoT Greengrass Core software runs a component.	June 3, 2021
AWS IoT Device Tester v4.0.2 with GGV2Q v1.1.0 released	Version 4.0.2 of IDT for AWS IoT Greengrass V2 is available . This release includes the AWS IoT Greengrass V2 q ualification suite (GGV2Q) v1.1.0 and supports using Greengrass nucleus v2.1.0 with Greengrass CLI v2.1.0 for device qualification. This also includes new required test groups for MQTT and Lambda, and other minor bug fixes and improvements.	May 5, 2021

Dockerfile and Docker images for AWS IoT Greengrass Core software v2.1.0	The Dockerfile and Docker image for AWS IoT Greengrass Core software v2.1.0 are now available. The Docker image enables you to run the AWS IoT Greengrass Core software in a Docker container that uses Amazon Linux 2 as the base operating system.	April 27, 2021
AWS IoT Greengrass Core v2.1.0 software update	This release provides version 2.1.0 of the Greengrass nucleus component and updates AWS-provided components. It includes a new component that you can use to download Docker images from private Amazon ECR repositories, and new sample components to perform machine learning inference using TensorFlow Lite.	April 26, 2021
Example component that uses Secrets Manager	Added an example component that prints the value of an AWS Secrets Manager secret that you deploy to a core device.	April 8, 2021
Minimal AWS IoT policy for Greengrass core devices	Added information about the minimal set of permissio ns required to support basic Greengrass functionality on a core device.	April 2, 2021

Subscribe to IPC event streams	Added information about how to use interprocess communication (IPC) operations to subscribe to streams of events on a Greengrass core device.	April 1, 2021
Support update for AWS IoT Device Tester for AWS IoT Greengrass	IDT for AWS IoT Greengrass V2 version 4.0.1 now supports using Greengrass nucleus version 2.0.5 with Greengrass CLI version 2.0.5 for device qualification.	March 17, 2021
Create custom components that use stream manager	Added information about how to configure component recipes and artifacts to develop applications that manage data streams.	March 9, 2021
AWS IoT Greengrass Core v2.0.5 software update	This release provides version 2.0.5 of the Greengrass nucleus component and updates AWS-provided components. It fixes an issue with network proxy support and an issue with the Greengrass data plane endpoint in AWS China Regions.	March 9, 2021

Component environment variable reference	Added information about the environment variables that the AWS IoT Greengrass Core software sets for components. You can use these environme nt variables to get the thing name, AWS Region, and Greengrass nucleus version.	February 23, 2021
Manual installation	Added information about how to create required AWS resources manually or to install behind a firewall or network proxy. By using a manual installation, you don't need to give the installer permission to create resources in your AWS account, because you create the the required AWS IoT and IAM resources. You can also configure your device to connect on port 443 or through a network proxy.	February 17, 2021
AWS IoT Greengrass Core IPC library update in AWS IoT Device SDK for Python v2	Version 1.5.4 of the AWS IoT Device SDK for Python v2 simplifies the steps required to connect to the AWS IoT Greengrass Core IPC service.	February 11, 2021
Support update for AWS IoT Device Tester for AWS IoT Greengrass	IDT for AWS IoT Greengrass V2 version 4.0.1 now supports using Greengrass nucleus version 2.0.4 with Greengra ss CLI version 2.0.4 for device qualification.	February 5, 2021

New tutorial to import Lambda functions	Added a new console-based tutorial to import a Lambda function as a component that runs on Greengrass core device.	February 5, 2021
AWS IoT Greengrass Core v2.0.4 software update	This release provides version 2.0.4 of the Greengras s nucleus component. It includes the new `greengras sDataPlanePort` parameter to configure HTTPS communication over port 443 and fixes bugs. The minimal IAM policy now requires the `iam:GetPo licy` and `sts:GetCa llerIdentity` when the AWS IoT Greengrass Core software installer is run with `--provision true` .	February 4, 2021
New secure tunneling component released	Version 1.0.0 of the secure tunneling component is available for AWS IoT Greengrass. This AWS-provi ded component uses AWS IoT secure tunneling to establish secure bidirecti onal communication with a Greengrass core device that is behind restricted firewalls.	January 21, 2021

AWS IoT Device Tester for AWS IoT Greengrass v4.0.1 released	Version 4.0.1 of IDT for AWS IoT Greengrass V2 is available . This version enables you to use IDT to develop and run your custom test suites for device validation. This also includes code signed IDT applications for macOS and Windows.	December 22, 2020
Initial release of AWS IoT Greengrass Version 2	AWS IoT Greengrass V2 is a new major version release of AWS IoT Greengrass. This version adds several features such as modular software components and continuous deployments. These features make it easier for you to develop and manage edge applications.	December 15, 2020

AWS Glossary

For the latest AWS terminology, see the AWS glossary in the *AWS Glossary Reference*.