Table of Contents

Getting Started with AWS IoT Greengrass ................................................................. 64
  AWS IoT Greengrass Core Software ........................................................................... 2
  AWS IoT Greengrass Core Versions .......................................................................... 2
  AWS IoT Greengrass Groups ..................................................................................... 6
  Devices in AWS IoT Greengrass .............................................................................. 8
  SDKs ........................................................................................................................... 9
  Supported Platforms and Requirements .................................................................. 11
  AWS IoT Greengrass Downloads ............................................................................. 16
  AWS IoT Greengrass Core ....................................................................................... 24
  AWS IoT Greengrass Core Configuration File .......................................................... 24
  Endpoints Must Match the Certificate Type ............................................................. 45
  Connect on Port 443 or Through a Network Proxy .................................................. 46
  Write Directory .......................................................................................................... 52
  Message Quality of Service ...................................................................................... 54
  MQTT Message Queue ............................................................................................. 55
  Client IDs for MQTT Connections with AWS IoT .................................................... 58
  Activate Automatic IP Detection ............................................................................ 59
  Start Greengrass on System Boot ........................................................................... 62
  Archive a Core Software Installation ...................................................................... 62
  See Also ..................................................................................................................... 63

Module 1: Environment Setup for Greengrass ............................................................ 65
  Setting Up a Raspberry Pi .......................................................................................... 65
  Setting Up an Amazon EC2 Instance ....................................................................... 70
  Setting Up Other Devices ......................................................................................... 75

Module 2: Installing the AWS IoT Greengrass Core Software ....................................... 76
  Configure AWS IoT Greengrass on AWS IoT .......................................................... 77
  Start AWS IoT Greengrass on the Core Device ....................................................... 81

Module 3 (Part 1): Lambda Functions on AWS IoT Greengrass ...................................... 84
  Create and Package a Lambda Function .................................................................. 84
  Configure the Lambda Function for AWS IoT Greengrass ....................................... 88
  Deploy Cloud Configurations to a Core Device ...................................................... 93
  Verify the Lambda Function Is Running on the Device ......................................... 94

Module 3 (Part 2): Lambda Functions on AWS IoT Greengrass ...................................... 97
  Create and Package the Lambda Function ................................................................ 97
  Configure Long-Lived Lambda Functions for AWS IoT Greengrass ....................... 99
  Test Long-Lived Lambda Functions ....................................................................... 100
  Test On-Demand Lambda Functions ..................................................................... 105

Module 4: Interacting with Devices in an AWS IoT Greengrass Group ............................ 109
  Create AWS IoT Devices in an AWS IoT Greengrass Group ................................... 110
  Configure Subscriptions ........................................................................................ 113
  Install the AWS IoT Device SDK for Python ........................................................ 114
  Test Communications ............................................................................................. 119

Module 5: Interacting with Device Shadows .................................................................. 122
  Configure Devices and Subscriptions ..................................................................... 123
<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Download Required Files</td>
</tr>
<tr>
<td>Test Communications (Device Syncs Disabled)</td>
</tr>
<tr>
<td>Test Communications (Device Syncs Enabled)</td>
</tr>
<tr>
<td>Module 6: Accessing Other AWS Services</td>
</tr>
<tr>
<td>Configure the Group Role</td>
</tr>
<tr>
<td>Create and Configure the Lambda Function</td>
</tr>
<tr>
<td>Configure Subscriptions</td>
</tr>
<tr>
<td>Test Communications</td>
</tr>
<tr>
<td>Module 7: Simulating Hardware Security Integration</td>
</tr>
<tr>
<td>Install SoftHSM</td>
</tr>
<tr>
<td>Configure SoftHSM</td>
</tr>
<tr>
<td>Import the Private Key</td>
</tr>
<tr>
<td>Configure the Greengrass Core</td>
</tr>
<tr>
<td>Test the Configuration</td>
</tr>
<tr>
<td>See Also</td>
</tr>
<tr>
<td>OTA Updates of AWS IoT Greengrass Core Software</td>
</tr>
<tr>
<td>Greengrass OTA Update Agent</td>
</tr>
<tr>
<td>Integration with init Systems</td>
</tr>
<tr>
<td>OTA Self-Update with Managed Respawn</td>
</tr>
<tr>
<td>AWS IoT Greengrass Core Update with Managed Respawn</td>
</tr>
<tr>
<td>OTA Update Agent Self-Update</td>
</tr>
<tr>
<td>Greengrass Core Software Update</td>
</tr>
<tr>
<td>Deploy AWS IoT Greengrass Groups</td>
</tr>
<tr>
<td>Deploying Groups from the AWS IoT console</td>
</tr>
<tr>
<td>Deploying Groups with the AWS IoT Greengrass API</td>
</tr>
<tr>
<td>Overview of the Group Object Model</td>
</tr>
<tr>
<td>Groups</td>
</tr>
<tr>
<td>Group Versions</td>
</tr>
<tr>
<td>Group Components</td>
</tr>
<tr>
<td>Updating Groups</td>
</tr>
<tr>
<td>See Also</td>
</tr>
<tr>
<td>Get Deployment Notifications</td>
</tr>
<tr>
<td>Group Deployment Status Change Event</td>
</tr>
<tr>
<td>Prerequisites for Creating EventBridge Rules</td>
</tr>
<tr>
<td>Configure Deployment Notifications (Console)</td>
</tr>
<tr>
<td>Configure Deployment Notifications (CLI)</td>
</tr>
<tr>
<td>Configure Deployment Notifications (AWS CloudFormation)</td>
</tr>
<tr>
<td>See Also</td>
</tr>
<tr>
<td>Reset Deployments</td>
</tr>
<tr>
<td>Reset Deployments from the AWS IoT console</td>
</tr>
<tr>
<td>Reset Deployments with the AWS IoT Greengrass API</td>
</tr>
<tr>
<td>See Also</td>
</tr>
<tr>
<td>Create Bulk Deployments</td>
</tr>
<tr>
<td>Prerequisites</td>
</tr>
<tr>
<td>Create and Upload the Bulk Deployment Input File</td>
</tr>
<tr>
<td>Create and Configure an IAM Execution Role</td>
</tr>
<tr>
<td>Allow Your Execution Role Access to Your S3 Bucket</td>
</tr>
<tr>
<td>Deploy the Groups</td>
</tr>
<tr>
<td>Test the Deployment</td>
</tr>
<tr>
<td>Troubleshooting Bulk Deployments</td>
</tr>
<tr>
<td>See Also</td>
</tr>
<tr>
<td>Run Local Lambda Functions</td>
</tr>
<tr>
<td>SDKs</td>
</tr>
<tr>
<td>Migrating Cloud-Based Lambda Functions</td>
</tr>
<tr>
<td>Function Aliases and Versions</td>
</tr>
<tr>
<td>Controlling Greengrass Lambda Function Execution</td>
</tr>
<tr>
<td>Running a Lambda Function as Root</td>
</tr>
<tr>
<td>Topic</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>Access Local Resources</td>
</tr>
<tr>
<td>Supported Resource Types</td>
</tr>
<tr>
<td>Requirements</td>
</tr>
<tr>
<td>Lifecycle Configuration</td>
</tr>
<tr>
<td>Retrieve the Input Topic (or Subject)</td>
</tr>
<tr>
<td>Prerequisites</td>
</tr>
<tr>
<td>Get the AWS IoT Greengrass Container Image from Amazon ECR</td>
</tr>
<tr>
<td>Create and Configure the Greengrass Group and Core</td>
</tr>
<tr>
<td>Run AWS IoT Greengrass Locally</td>
</tr>
<tr>
<td>Configure &quot;No container&quot; Containerization for the Group</td>
</tr>
<tr>
<td>Deploy Lambda Functions to the Docker Container</td>
</tr>
<tr>
<td>(Optional) Deploy Devices that Interact with Greengrass in the Docker Container</td>
</tr>
<tr>
<td>Stopping the AWS IoT Greengrass Docker Container</td>
</tr>
<tr>
<td>Troubleshooting AWS IoT Greengrass in a Docker Container</td>
</tr>
<tr>
<td>Using the Console</td>
</tr>
<tr>
<td>Prerequisites</td>
</tr>
<tr>
<td>Create a Lambda Function Deployment Package</td>
</tr>
<tr>
<td>Create and Publish a Lambda Function</td>
</tr>
<tr>
<td>Add the Lambda Function to the Group</td>
</tr>
<tr>
<td>Troubleshooting</td>
</tr>
<tr>
<td>Using the CLI</td>
</tr>
<tr>
<td>Create Local Resources</td>
</tr>
<tr>
<td>Create the Greengrass Function</td>
</tr>
<tr>
<td>Add the Lambda Function to the Group</td>
</tr>
<tr>
<td>Troubleshooting</td>
</tr>
<tr>
<td>Perform Machine Learning Inference</td>
</tr>
<tr>
<td>How AWS IoT Greengrass ML Inference Works</td>
</tr>
<tr>
<td>Machine Learning Resources</td>
</tr>
<tr>
<td>Supported Model Sources</td>
</tr>
<tr>
<td>Requirements</td>
</tr>
<tr>
<td>Runtimes and Precompiled Framework Libraries for ML Inference</td>
</tr>
<tr>
<td>Amazon SageMaker Neo Deep Learning Runtime</td>
</tr>
<tr>
<td>MXNet Versioning</td>
</tr>
<tr>
<td>MXNet on Raspberry Pi</td>
</tr>
<tr>
<td>TensorFlow Model-Serving Limitations on Raspberry Pi</td>
</tr>
<tr>
<td>How to Configure Machine Learning Inference</td>
</tr>
<tr>
<td>Prerequisites</td>
</tr>
<tr>
<td>Configure the Raspberry Pi</td>
</tr>
<tr>
<td>Install the MXNet Framework</td>
</tr>
<tr>
<td>Create a Model Package</td>
</tr>
<tr>
<td>Create and Publish a Lambda Function</td>
</tr>
<tr>
<td>Add the Lambda Function to the Group</td>
</tr>
<tr>
<td>Topic</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>Using AWS IoT Device Tester for AWS IoT Greengrass</td>
</tr>
<tr>
<td>Tagging Your Greengrass Resources</td>
</tr>
<tr>
<td>Monitoring</td>
</tr>
<tr>
<td>CloudWatch Logs</td>
</tr>
<tr>
<td>File System Logs</td>
</tr>
<tr>
<td>Default Logging Configuration</td>
</tr>
<tr>
<td>Configure Logging for AWS IoT Greengrass</td>
</tr>
<tr>
<td>Configure Logging (Console)</td>
</tr>
<tr>
<td>Configure Logging (API)</td>
</tr>
<tr>
<td>Configuration Example</td>
</tr>
<tr>
<td>Logging Limitations</td>
</tr>
<tr>
<td>Transactions per Second</td>
</tr>
<tr>
<td>Memory</td>
</tr>
<tr>
<td>Clock Skew</td>
</tr>
<tr>
<td>Disk Usage</td>
</tr>
<tr>
<td>Log Loss</td>
</tr>
<tr>
<td>Logging AWS IoT Greengrass API Calls with AWS CloudTrail</td>
</tr>
<tr>
<td>AWS IoT Greengrass Information in CloudTrail</td>
</tr>
<tr>
<td>Understanding AWS IoT Greengrass Log File Entries</td>
</tr>
<tr>
<td>See Also</td>
</tr>
<tr>
<td>CloudTrail Logs</td>
</tr>
<tr>
<td>Tagging Support (Console)</td>
</tr>
<tr>
<td>Tagging Support (API)</td>
</tr>
<tr>
<td>Using Tags with IAM Policies</td>
</tr>
<tr>
<td>Example IAM Policies</td>
</tr>
<tr>
<td>See Also</td>
</tr>
<tr>
<td>AWS CloudFormation Support for AWS IoT Greengrass</td>
</tr>
<tr>
<td>Create Resources</td>
</tr>
<tr>
<td>Deploy Resources</td>
</tr>
<tr>
<td>Example Template</td>
</tr>
<tr>
<td>Supported AWS Regions</td>
</tr>
<tr>
<td>Using AWS IoT Device Tester for AWS IoT Greengrass</td>
</tr>
<tr>
<td>AWS IoT Device Tester for AWS IoT Greengrass Versions</td>
</tr>
<tr>
<td>Earlier IDT Versions for AWS IoT Greengrass</td>
</tr>
<tr>
<td>Prerequisites</td>
</tr>
<tr>
<td>Download the Latest Version of AWS IoT Device Tester for AWS IoT Greengrass</td>
</tr>
<tr>
<td>Create and Configure an AWS Account</td>
</tr>
<tr>
<td>Using IDT for Greengrass 1.6.1 and Earlier</td>
</tr>
<tr>
<td>Configure Your Device</td>
</tr>
<tr>
<td>Setting Configuration to Run the AWS IoT Greengrass Qualification Suite</td>
</tr>
<tr>
<td>Configure Your AWS Credentials</td>
</tr>
<tr>
<td>Device Configuration</td>
</tr>
<tr>
<td>Running Tests</td>
</tr>
<tr>
<td>AWS IoT Device Tester for AWS IoT Greengrass Commands</td>
</tr>
<tr>
<td>Understanding Results and Logs</td>
</tr>
<tr>
<td>Viewing Results</td>
</tr>
</tbody>
</table>
Troubleshooting ............................................................................................................................. 516
Deployment Issues .................................................................................................................. 524
AWS IoT Greengrass Core Issues ............................................................................................... 516
IDT for AWS IoT Greengrass Troubleshooting ............................................................................. 504
Error Codes ............................................................................................................................. 504
Resolving IDT for AWS IoT Greengrass Errors ........................................................................ 513
Troubleshooting ....................................................................................................................... 516
AWS IoT Greengrass Core Issues ............................................................................................... 516

Error: The configuration file is missing the CaPath, CertPath or KeyPath. The Greengrass
d daemon process with [pid = <pid>] died. ................................................................. 517
Error: Failed to parse <greengrass-root>/config/config.json. ........................................ 518
Error: Error occurred while generating TLS config. ErrUnknownURIScheme ...................... 518
Error: Runtime failed to start: unable to start workers: container test timed out. .............. 518
Error: Failed to invoke PutLogEvents on local Cloudwatch, logGroup:/GreengrassSystem/
connection_manager, error: RequestError: send request failed caused by: Post http://<path>/
cloudwatch/logs/: dial tcp <address>: getsockopt: connection refused, response: {} ... 519
Error: Unable to create server due to: failed to load group: chmod <greengrass-root>/ggc/
deployment/lambda/arm:aws:lambda:<region>:<account-id>:function:<function-name>:<version>/<file-name>: no such file or directory. ........................................ 519
The AWS IoT Greengrass Core software doesn't start after you changed from running with no
cointerization to running in a Greengrass container. .................................................. 519
Error: Spool size should be at least 262144 bytes. .............................................................. 520
Error: container_linux.go:344: starting container process caused "process_linux.go:424:
container init caused \"roofs_linux.go:64: mounting \"/greengrass/ggc/socket/
greengrass_ipc.sock\" to roots \"/greengrass/ggc/packages/<version>/roofs/merged\"\"
at \"/greengrass_ipc.sock\" caused \"stat /greengrass/ggc/socket/greengrass_ipc.sock:
permission denied\"\"\"\". .................................................. 520
Error: Greengrass daemon running with PID: <process-id>. Some system components failed to
tart. Check ‘runtime.log’ for errors. ............................................................................ 520
Device shadow does not sync with the cloud. ...................................................................... 520
The AWS IoT Greengrass Core software does not run on Raspberry Pi because user namespace
is not enabled. ................................................................................................................. 521
ERROR: unable to accept TCP connection. accept tcp [::]:8000: accept4: too many open files. .... 521
Error: Runtime execution error: unable to start lambda container. container_linux.go:259:
starting container process caused "process_linux.go:345: container init caused
\"roofs_linux.go:50: preparing roots caused \"permission denied\"\"\"\". .......................... 521
Warning: [WARN]-[S]GK Remote: Error retrieving public key data: ErrPrincipalNotConfigured:
private key for MqttCertificate is not set. ................................................................. 522
Error: Permission denied when attempting to use role arn:aws:iam::<account-id>:role/<role-
name> to access s3 url https://<region>-greengrass-updates.s3.<region>.amazonaws.com/
core/<architecture>/greengrass-core-<distribution-version>.tar.gz. .............................. 522
The AWS IoT Greengrass core is configured to use a network proxy and your Lambda function
can't make outgoing connections. ................................................................................ 522
The core is in an infinite connect-disconnect loop. The runtime.log file contains a continuous
series of connect and disconnect entries. ........................................................................... 523
Error: unable to start lambda container. container_linux.go:259: starting container process
\" to roots \"\"\". .................................................................. 523
Error: [ERROR]-runtime execution error: unable to start lambda container. {"errorString":
"failed to initialize container mounts: failed to create overlay fs for container: mounting overlay at
greengrass/ggc/packages/<ggc-version>/roofs/merged failed: failed to mount with
args source=\"no_source\" dest=\"/greengrass/ggc/packages/<ggc-version>/roofs/merged
\" fstype=\"overlay\" flags=\"0\" data=\"lowerdir=greengrass/ggc/packages/<ggc-version>/
dns/upperdir=greengrass/ggc/packages/<ggc-version>/roofs/workdir=greengrass/ggc/packages/<ggc-
version>/roofs/work\": too many levels of symbolic links\"} ................................. 524
Error: [DEBUG]-Failed to get routes. Discarding message. ................................................ 524
Deployment Issues .............................................................................................................. 524
Your current deployment does not work and you want to revert to a previous working
development. ............................................................................................................... 525
You see a 403 Forbidden error on deployment in the logs. ................................................................. 527
A ConcurrentDeployment error occurs when you run the create-deployment command for the first time. ................................................................. 527
Error: Greengrass is not authorized to assume the Service Role associated with this account, or the error: Failed: TES service role is not associated with this account. ........................................ 527
The deployment doesn't finish. .................................................................................................. 527
The deployment doesn't finish, and runtime.log contains multiple "wait 1s for container to stop" entries. ......................................................................................................... 528
Error: Deployment <deployment-id> of type NewDeployment for group <group-id> failed error: Error while processing. group config is invalid: 112 or [119 0] don't have rw permission on the file: <path>. ........................................................................................................ 528
Error: <list-of-function-arms> are configured to run as root but Greengrass is not configured to run Lambda functions with root permissions. .................................................. 528
Error: Deployment <deployment-id> of type NewDeployment for group <group-id> failed error: Greengrass deployment error: unable to execute download step in deployment. error while processing: unable to load the group file downloaded: could not find UID based on user name, userName: ggc_user: user: unknown user ggc_user. .......................................................... 529
Error: [WARN]-MQTT[client] dial tcp: lookup <host-prefix>-ats.iot.<region>.amazonaws.com: no such host ... [ERROR]-Greengrass deployment error: failed to report deployment status back to cloud ... net/http: request canceled while waiting for connection (Client.Timeout exceeded while awaiting headers) .................................................................................................. 529
Create Group/Create Function Issues .......................................................................................... 530
Error: Your 'IsolationMode' configuration for the group is invalid. .............................................. 530
Error: Your 'IsolationMode' configuration for function with arn <function-arn> is invalid. .......... 531
Error: MemorySize configuration for function with arn <function-arn> is not allowed in IsolationMode=NoContainer. ................................................................. 531
Error: Access Sysfs configuration for function with arn <function-arn> is not allowed in IsolationMode=NoContainer. ................................................................. 531
Error: MemorySize configuration for function with arn <function-arn> is required in IsolationMode=GreengrassContainer. ................................................................. 531
Error: Function <function-arn> refers to resource of type <resource-type> that is not allowed in IsolationMode=NoContainer. ................................................................. 531
Error: Execution configuration for function with arn <function-arn> is not allowed. ................ 532
Discovery Issues ....................................................................................................................... 532
Error: Device is a member of too many groups, devices may not be in more than 10 groups ..... 532
Error: Unknown options: -no-include-email ........................................................................ 198
Warning: IPv4 is disabled. Networking will not work. ............................................................... 198
Error: A firewall is blocking file Sharing between windows and the containers. ....................... 198
Error: Cannot create container for the service greengrass: Conflict. The container name "/aws-iot-greengrass" is already in use. ................................................................. 533
Error: [FATAL]-Failed to reset thread's mount namespace due to an unexpected error: "operation not permitted". To maintain consistency, GGC will crash and need to be manually restarted. ................................................................. 533
Troubleshooting with Logs ........................................................................................................ 534
Troubleshooting Storage Issues .................................................................................................. 534
Troubleshooting Messages .......................................................................................................... 535
Troubleshooting Shadow Synchronization Timeout Issues ........................................................ 535
Check the AWS IoT Greengrass Forum ....................................................................................... 536
Document History .......................................................................................................................... 537
Earlier Updates ............................................................................................................................ 541
What Is AWS IoT Greengrass?

AWS IoT Greengrass is software that extends cloud capabilities to local devices. This enables devices to collect and analyze data closer to the source of information, react autonomously to local events, and communicate securely with each other on local networks. AWS IoT Greengrass developers can use AWS Lambda functions and prebuilt connectors to create serverless applications that are deployed to devices for local execution.

The following diagram shows the basic architecture of AWS IoT Greengrass.

AWS IoT Greengrass makes it possible for customers to build IoT devices and application logic. Specifically, AWS IoT Greengrass provides cloud-based management of application logic that runs on devices. Locally deployed Lambda functions and connectors are triggered by local events, messages from the cloud, or other sources.

In AWS IoT Greengrass, devices securely communicate on a local network and exchange messages with each other without having to connect to the cloud. AWS IoT Greengrass provides a local pub/sub message manager that can intelligently buffer messages if connectivity is lost so that inbound and outbound messages to the cloud are preserved.

AWS IoT Greengrass protects user data:

- Through the secure authentication and authorization of devices.
- Through secure connectivity in the local network.
- Between local devices and the cloud.

Device security credentials function in a group until they are revoked, even if connectivity to the cloud is disrupted, so that the devices can continue to securely communicate locally.

AWS IoT Greengrass provides secure, over-the-air software updates of Lambda functions.

AWS IoT Greengrass consists of:

- Software distributions
  - AWS IoT Greengrass Core software
  - AWS IoT Greengrass core SDK
- Cloud service
  - AWS IoT Greengrass API
• Features
  • Lambda runtime
  • Shadows implementation
  • Message manager
  • Group management
  • Discovery service
  • Over-the-air update agent
  • Local resource access
  • Local machine learning inference
  • Local secrets manager
  • Connectors with built-in integration with services, protocols, and software

AWS IoT Greengrass Core Software

The AWS IoT Greengrass Core software provides the following functionality:

• Deployment and local execution of connectors and Lambda functions.
• Secure, encrypted storage of local secrets and controlled access by connectors and Lambda functions.
• MQTT messaging over the local network between devices, connectors, and Lambda functions using managed subscriptions.
• MQTT messaging between AWS IoT and devices, connectors, and Lambda functions using managed subscriptions.
• Secure connections between devices and the cloud using device authentication and authorization.
• Local shadow synchronization of devices. Shadows can be configured to sync with the cloud.
• Controlled access to local device and volume resources.
• Deployment of cloud-trained machine learning models for running local inference.
• Automatic IP address detection that enables devices to discover the Greengrass core device.
• Central deployment of new or updated group configuration. After the configuration data is downloaded, the core device is restarted automatically.
• Secure, over-the-air software updates of user-defined Lambda functions.

AWS IoT Greengrass core instances are configured through AWS IoT Greengrass APIs that create and update AWS IoT Greengrass group definitions stored in the cloud.

AWS IoT Greengrass Core Versions

The following tabs describe what's new and changed in AWS IoT Greengrass Core software versions.

GGC v1.9

1.9.4 - Current version

  Bug fixes and improvements:
  • General performance improvements and bug fixes.

1.9.3

New features:
• Support for Armv6l. AWS IoT Greengrass Core software v1.9.3 or later can be installed on Raspbian distributions on Armv6l architectures (for example, on Raspberry Pi Zero devices).
• OTA updates on port 443 with ALPN. Greengrass cores that use port 443 for MQTT traffic now support over-the-air (OTA) software updates. AWS IoT Greengrass uses the Application Layer Protocol Network (ALPN) TLS extension to enable these connections. For more information, see [OTA Updates of AWS IoT Greengrass Core Software](p. 148) and the section called “Connect on Port 443 or Through a Network Proxy” (p. 46).

Bug fixes and improvements:
• Fixes a bug introduced in v1.9.0 that prevented Python 2.7 Lambda functions from sending binary payloads to other Lambda functions.
• General performance improvements and bug fixes.

1.9.2

New features:
• Support for OpenWrt. AWS IoT Greengrass Core software v1.9.2 or later can be installed on OpenWrt distributions with Armv8 (AArch64) and Armv7l architectures. Currently, OpenWrt does not support ML inference.

1.9.1

Bug fixes and improvements:
• Fixes a bug introduced in v1.9.0 that drops messages from the cloud that contain wildcard characters in the topic.

1.9.0

New features:
• Support for Python 3.7 and Node.js 8.10 Lambda runtimes. Lambda functions that use Python 3.7 and Node.js 8.10 runtimes can now run on an AWS IoT Greengrass core. (AWS IoT Greengrass continues to support the Python 2.7 and Node.js 6.10 runtimes.)
• Optimized MQTT connections. The Greengrass core establishes fewer connections with the AWS IoT Core. This change can reduce operational costs for charges that are based on the number of connections.
• Elliptic Curve (EC) key for the local MQTT server. The local MQTT server supports EC keys in addition to RSA keys. (The MQTT server certificate has an SHA-256 RSA signature, regardless of the key type.) For more information, see the section called “Security Principals” (p. 439).

Bug fixes and improvements:
• General performance improvements and bug fixes.

GGC v1.8

1.8.4

Fixed an issue with shadow synchronization and device certificate manager reconnection.

General performance improvements and bug fixes.

1.8.3

General performance improvements and bug fixes.

1.8.2

General performance improvements and bug fixes.

1.8.1

General performance improvements and bug fixes.

1.8.0

New features:
AWS IoT Greengrass Developer Guide
AWS IoT Greengrass Core Versions

- Configurable default access identity for Lambda functions in the group. This group-level setting determines the default permissions that are used to run Lambda functions. You can set the user ID, group ID, or both. Individual Lambda functions can override the default access identity of their group. For more information, see the section called “Setting the Default Access Identity for Lambda Functions in a Group” (p. 183).

- HTTPS traffic over port 443. HTTPS communication can be configured to travel over port 443 instead of the default port 8443. This complements AWS IoT Greengrass support for the Application Layer Protocol Network (ALPN) TLS extension and allows all Greengrass messaging traffic—both MQTT and HTTPS—to use port 443. For more information, see the section called “Connect on Port 443 or Through a Network Proxy” (p. 46).

- Predictably named client IDs for AWS IoT connections. This change enables support for AWS IoT Device Defender and AWS IoT Lifecycle events, so you can receive notifications for connect, disconnect, subscribe, and unsubscribe events. Predictable naming also makes it easier to create logic around connection IDs (for example, to create subscribe policy templates based on certificate attributes). For more information, see the section called “Client IDs for MQTT Connections with AWS IoT” (p. 58).

Bug fixes and improvements:
- Fixed an issue with shadow synchronization and device certificate manager reconnection.
- General performance improvements and bug fixes.

GGC v1.7

1.7.1

New features:
- Greengrass connectors provide built-in integration with local infrastructure, device protocols, AWS, and other cloud services. For more information, see Integrate with Services and Protocols Using Connectors (p. 283).
- AWS IoT Greengrass extends AWS Secrets Manager to core devices, which makes your passwords, tokens, and other secrets available to connectors and Lambda functions. Secrets are encrypted in transit and at rest. For more information, see Deploy Secrets to the Core (p. 263).
- Support for a hardware root of trust security option. For more information, see the section called “Hardware Security” (p. 448).
- Isolation and permission settings that allow Lambda functions to run without Greengrass containers and to use the permissions of a specified user and group. For more information, see the section called “Controlling Greengrass Lambda Function Execution” (p. 178).
- You can run AWS IoT Greengrass in a Docker container (on Windows, macOS, or Linux) by configuring your Greengrass group to run with no containerization. For more information, see the section called “Run AWS IoT Greengrass in a Docker Container” (p. 189).
- MQTT messaging on port 443 with Application Layer Protocol Negotiation (ALPN) or connection through a network proxy. For more information, see the section called “Connect on Port 443 or Through a Network Proxy” (p. 46).
- The Amazon SageMaker Neo deep learning runtime, which supports machine learning models that have been optimized by the Amazon SageMaker Neo deep learning compiler. For information about the Neo deep learning runtime, see the section called “Runtimes and Precompiled Framework Libraries for ML Inference” (p. 224).

Bug fixes and improvements:
- General performance improvements and bug fixes.

In addition, the following features are available with this release:
• The AWS IoT Device Tester for AWS IoT Greengrass, which you can use to verify that your 
CPU architecture, kernel configuration, and drivers work with AWS IoT Greengrass. For more 
information, see Using AWS IoT Device Tester for AWS IoT Greengrass (p. 482).

• The AWS IoT Greengrass Core software, AWS IoT Greengrass Core SDK, and AWS 
IoT Greengrass Machine Learning SDK packages are available for download through 
Amazon CloudFront. For more information, see the section called “AWS IoT Greengrass 
Downloads” (p. 16).

Deprecated versions

1.6.1

New features:

• Lambda executables that run binary code on the Greengrass core. Use the new AWS IoT 
Greengrass Core SDK for C to write Lambda executables in C and C++. For more information, 
see the section called “Lambda Executables” (p. 187).

• Optional local storage message cache that can persist across restarts. You can configure the 
storage settings for MQTT messages that are queued for processing. For more information, 
see the section called “MQTT Message Queue” (p. 55).

• Configurable maximum reconnect retry interval for when the core device is disconnected. For 
more information, see the mqttMaxConnectionRetryInterval property in the section 
called “AWS IoT Greengrass Core Configuration File” (p. 24).

• Local resource access to the host /proc directory. For more information, see Access Local 
Resources (p. 200).

• Configurable write directory. The AWS IoT Greengrass Core software can be deployed to 
read-only and read-write locations. For more information, see the section called “Write 
Directory” (p. 52).

Bug fixes and improvements:

• Performance improvement for publishing messages in the Greengrass core and between 
devices and the core.

• Reduced the compute resources required to process logs generated by user-defined Lambda 
functions.

1.5.0

New features:

• AWS IoT Greengrass Machine Learning (ML) Inference is generally available. You can perform 
ML inference locally on AWS IoT Greengrass devices using models that are built and trained in 
the cloud. For more information, see Perform Machine Learning Inference (p. 221).

• Greengrass Lambda functions now support binary data as input payload, in addition to JSON. 
To use this feature, you must upgrade to AWS IoT Greengrass Core SDK version 1.1.0, which 
you can download from the AWS IoT Greengrass Core SDK (p. 22) downloads page.

Bug fixes and improvements:

• Reduced the overall memory footprint.

• Performance improvements for sending messages to the cloud.

• Performance and stability improvements for the download agent, Device Certificate Manager, 
and OTA update agent.

• Minor bug fixes.

1.3.0

New features:
• Over-the-air (OTA) update agent capable of handling cloud-deployed, Greengrass update jobs. The agent is found under the new /greengrass/ota directory. For more information, see OTA Updates of AWS IoT Greengrass Core Software (p. 148).

• Local resource access feature allows Greengrass Lambda functions to access local resources, such as peripheral devices and volumes. For more information, see Access Local Resources with Lambda Functions and Connectors (p. 200).

1.1.0

New features:

• Deployed AWS IoT Greengrass groups can be reset by deleting Lambda functions, subscriptions, and configurations. For more information, see the section called “Reset Deployments” (p. 163).

• Support for Node.js 6.10 and Java 8 Lambda runtimes, in addition to Python 2.7.

To migrate from the previous version of the AWS IoT Greengrass core:

• Copy certificates from the /greengrass/configuration/certs folder to /greengrass/certs.

• Copy /greengrass/configuration/config.json to /greengrass/config/config.json.

• Run /greengrass/ggc/core/greengrassd instead of /greengrass/greengrassd.

• Deploy the group to the new core.

1.0.0

Initial version

You can download recent versions of the AWS IoT Greengrass Core software from the section called “AWS IoT Greengrass Core Software” (p. 16).

AWS IoT Greengrass Groups

An AWS IoT Greengrass group is a collection of settings and components, such as an AWS IoT Greengrass core, devices, and subscriptions. Groups are used to define a scope of interaction. For example, a group might represent one floor of a building, one truck, or an entire mining site. The following diagram shows the components that can make up an Greengrass group.
In the preceding diagram:

A: AWS IoT Greengrass group definition

A collection of information about the AWS IoT Greengrass group.

B: AWS IoT Greengrass group settings

These include:
- AWS IoT Greengrass group role.
- Certification authority and local connection configuration.
- AWS IoT Greengrass core connectivity information.
- Default Lambda runtime environment. For more information, see the section called “Setting Default Containerization for Lambda Functions in a Group” (p. 184).
- CloudWatch and local logs configuration. For more information, see Monitoring (p. 457).

C: AWS IoT Greengrass core

The AWS IoT thing that represents the AWS IoT Greengrass core. For more information, see the section called “AWS IoT Greengrass Core” (p. 24).

D: Lambda function definition

A list of Lambda functions that run locally on the core, with associated configuration data. For more information, see Run Local Lambda Functions (p. 175).

E: Subscription definition

A list of subscriptions that enable communication using MQTT messages. A subscription defines:
- A message source and message target. These can be devices, Lambda functions, connectors, AWS IoT, and the local shadow service
- A subject, which is an MQTT topic or topic filter that's used to filter message data.

For more information, see the section called “Greengrass Messaging Workflow” (p. 441).
F: Connector definition

A list of connectors that run locally on the core, with associated configuration data. For more information, see *Integrate with Services and Protocols Using Connectors (p. 283).*

G: Device definition

A list of AWS IoT things (devices) that are members of the AWS IoT Greengrass group, with associated configuration data. For more information, see the section called “Devices in AWS IoT Greengrass” (p. 8).

H: Resource definition

A list of local resources, machine learning resources, and secret resources on the AWS IoT Greengrass core, with associated configuration data. For more information, see *Access Local Resources (p. 200), Perform Machine Learning Inference (p. 221), and Deploy Secrets to the Core (p. 263).*

When deployed, the AWS IoT Greengrass group definition, Lambda functions, connectors, resources, and subscription table are copied to an AWS IoT Greengrass core device.

## Devices in AWS IoT Greengrass

An AWS IoT Greengrass group can contain two types of device:

**AWS IoT Greengrass core**

A core is an AWS IoT device that runs the AWS IoT Greengrass Core software, which enables it to communicate directly with the AWS IoT and AWS IoT Greengrass cloud services. A core has its own certificate used for authenticating with AWS IoT. It has a device shadow and exists in the AWS IoT device registry. AWS IoT Greengrass cores run a local Lambda runtime, a deployment agent, and an IP address tracker that sends IP address information to the AWS IoT Greengrass cloud service to allow Greengrass devices to automatically discover their group and core connection information. For more information, see the section called “AWS IoT Greengrass Core” (p. 24).

*Note*

A Greengrass group must contain exactly one core.

**AWS IoT device connected to a Greengrass core**

Connected devices (also called *Greengrass devices*) can connect to a core in the same Greengrass group. Greengrass devices run Amazon FreeRTOS or use the *AWS IoT Device SDK (p. 10)* or *AWS IoT Greengrass Discovery API (p. 426)* to get the connection information for the core. Devices have their own certificate for authentication, device shadow, and entry in the AWS IoT device registry. For more information, see the section called “Module 4: Interacting with Devices in an AWS IoT Greengrass Group” (p. 109).

In a Greengrass group, you can create subscriptions that allow devices to communicate over MQTT with Lambda functions, connectors, and other devices in the group, and with AWS IoT or the local shadow service. MQTT messages are routed through the core. If the core device loses connectivity to the cloud, devices can continue to communicate over the local network. Devices can vary in size, from smaller microcontroller-based devices to large appliances.

*Note*

Currently, a Greengrass group can contain up to 200 devices. A device can be a member of up to 10 groups.

The following table shows how these device types are related.
The AWS IoT Greengrass core device stores certificates in two locations:

- Core device certificate in /greengrass/certs - The core device certificate is named `hash.cert.pem` (for example, 86c84488a5.cert.pem). This certificate is used to authenticate the core when connecting to the AWS IoT and AWS IoT Greengrass services.
- MQTT core server certificate in /greengrass-root/ggc/var/state/server - The MQTT core server certificate is named `server.crt`. This certificate is used for mutual authentication between the local MQTT service (that's on the Greengrass core) and Greengrass devices before messages are exchanged.

### SDKs

The following AWS-provided SDKs are used to work with AWS IoT Greengrass:

**GGC 1.9**

**AWS SDK**

Use the AWS SDK to build applications that interact with any AWS service, including Amazon S3, Amazon DynamoDB, AWS IoT, AWS IoT Greengrass, and more. In the context of AWS IoT Greengrass, you can use the AWS SDK in deployed Lambda functions to make direct calls to any AWS service. For more information, see [AWS SDKs](#) (p. 176).
AWS IoT Device SDK

The AWS IoT Device SDK helps devices connect to AWS IoT or AWS IoT Greengrass services. Devices must know which AWS IoT Greengrass group they belong to and the IP address of the AWS IoT Greengrass core that they should connect to.

Although you can use any of the AWS IoT Device SDK platforms to connect to an AWS IoT Greengrass core, only the C++ and Python SDKs provide AWS IoT Greengrass-specific functionality, such as access to the AWS IoT Greengrass Discovery Service and AWS IoT Greengrass core root CA downloads. For more information, see AWS IoT Device SDK.

AWS IoT Greengrass Core SDK

The AWS IoT Greengrass Core SDK enables Lambda functions to interact with the Greengrass core, publish messages to AWS IoT, interact with the local shadow service, invoke other deployed Lambda functions, and access secret resources. This SDK is used by Lambda functions that run on an AWS IoT Greengrass core. For more information, see AWS IoT Greengrass Core SDK (p. 175).

AWS IoT Greengrass Machine Learning SDK

The AWS IoT Greengrass Machine Learning SDK enables Lambda functions to consume machine learning models that are deployed to the Greengrass core as machine learning resources. This SDK is used by Lambda functions that run on an AWS IoT Greengrass core and interact with a local inference service. For more information, see AWS IoT Greengrass Machine Learning SDK (p. 176).

GGC 1.8

AWS SDK

Use the AWS SDK to build applications that interact with any AWS service, including Amazon S3, Amazon DynamoDB, AWS IoT, AWS IoT Greengrass, and more. In the context of AWS IoT Greengrass, you can use the AWS SDK in deployed Lambda functions to make direct calls to any AWS service. For more information, see AWS SDKs (p. 176).

AWS IoT Device SDK

The AWS IoT Device SDK helps devices connect to AWS IoT or AWS IoT Greengrass services. Devices must know which AWS IoT Greengrass group they belong to and the IP address of the AWS IoT Greengrass core that they should connect to.

Although you can use any of the AWS IoT Device SDK platforms to connect to an AWS IoT Greengrass core, only the C++ and Python SDKs provide AWS IoT Greengrass-specific functionality, such as access to the AWS IoT Greengrass Discovery Service and AWS IoT Greengrass core root CA downloads. For more information, see AWS IoT Device SDK.

AWS IoT Greengrass Core SDK

The AWS IoT Greengrass Core SDK enables Lambda functions to interact with the Greengrass core, publish messages to AWS IoT, interact with the local shadow service, invoke other deployed Lambda functions, and access secret resources. This SDK is used by Lambda functions that run on an AWS IoT Greengrass core. For more information, see AWS IoT Greengrass Core SDK (p. 175).

AWS IoT Greengrass Machine Learning SDK

The AWS IoT Greengrass Machine Learning SDK enables Lambda functions to consume machine learning models that are deployed to the Greengrass core as machine learning resources. This SDK is used by Lambda functions that run on an AWS IoT Greengrass core and interact with
a local inference service. For more information, see AWS IoT Greengrass Machine Learning SDK (p. 176).

GGC 1.7

AWS SDK

Use the AWS SDK to build applications that interact with any AWS service, including Amazon S3, Amazon DynamoDB, AWS IoT, AWS IoT Greengrass, and more. In the context of AWS IoT Greengrass, you can use the AWS SDK in deployed Lambda functions to make direct calls to any AWS service. For more information, see AWS SDKs (p. 176).

AWS IoT Device SDK

The AWS IoT Device SDK helps devices connect to AWS IoT or AWS IoT Greengrass services. Devices must know which AWS IoT Greengrass group they belong to and the IP address of the AWS IoT Greengrass core that they should connect to.

Although you can use any of the AWS IoT Device SDK platforms to connect to an AWS IoT Greengrass core, only the C++ and Python SDKs provide AWS IoT Greengrass-specific functionality, such as access to the AWS IoT Greengrass Discovery Service and AWS IoT Greengrass core root CA downloads. For more information, see AWS IoT Device SDK.

AWS IoT Greengrass Core SDK

The AWS IoT Greengrass Core SDK enables Lambda functions to interact with the Greengrass core, publish messages to AWS IoT, interact with the local shadow service, invoke other deployed Lambda functions, and access secret resources. This SDK is used by Lambda functions that run on an AWS IoT Greengrass core. For more information, see AWS IoT Greengrass Core SDK (p. 175).

AWS IoT Greengrass Machine Learning SDK

The AWS IoT Greengrass Machine Learning SDK enables Lambda functions to consume machine learning models that are deployed to the Greengrass core as machine learning resources. This SDK is used by Lambda functions that run on an AWS IoT Greengrass core and interact with a local inference service. For more information, see AWS IoT Greengrass Machine Learning SDK (p. 176).

Supported Platforms and Requirements

The following tabs list supported platforms and requirements for the AWS IoT Greengrass Core software.

Note
You can download the AWS IoT Greengrass Core software from the AWS IoT Greengrass Core Software (p. 16) downloads.

GGC v1.9

Supported platforms:

- Architecture: Armv7l
- OS: Linux; Distribution: Raspbian Buster, 2019-07-10. AWS IoT Greengrass might work with other distributions for a Raspberry Pi, but we recommend Raspbian because it's the officially supported distribution.
- OS: Linux; Distribution: OpenWrt
- Architecture: Armv8 (AArch64)
Supported Platforms and Requirements

- OS: Linux; Distribution: Arch Linux
- OS: Linux; Distribution: OpenWrt
- Architecture: Armv6l
- OS: Linux; Distribution: Raspbian Buster, 2019-07-10
- Architecture: x86_64
- OS: Linux; Distribution: Amazon Linux (amzn2-ami-hvm-2.0.20190313-x86_64-gp2), Ubuntu 18.04

Windows, macOS, and Linux platforms can run AWS IoT Greengrass in a Docker container. For more information, see the section called “Run AWS IoT Greengrass in a Docker Container” (p. 189).

Requirements:

- Minimum 128 MB RAM allocated to the AWS IoT Greengrass core device.
- Linux kernel version:
  - Linux kernel version 4.4 or later is required to support running AWS IoT Greengrass with containers (p. 181).
  - Linux kernel version 3.17 or later is required to support running AWS IoT Greengrass without containers. In this configuration, the default Lambda function containerization for the Greengrass group must be set to No container. For instructions, see the section called “Setting Default Containerization for Lambda Functions in a Group” (p. 184).
- GNU C Library (glibc) version 2.14 or later. OpenWrt distributions require musl C Library version 1.1.16 or later.
- The /var/run directory must be present on the device.
- The /dev/stdin, /dev/stdout, and /dev/stderr files must be available.
- Hardlink and softlink protection must be enabled on the device. Otherwise, AWS IoT Greengrass can only be run in insecure mode, using the -i flag.
- The following Linux kernel configurations must be enabled on the device:
  - Namespace:
    - CONFIG_IPC_NS
    - CONFIG_UTS_NS
    - CONFIG_USER_NS
    - CONFIG_PID_NS
  - Cgroups:
    - CONFIG_CGROUP_DEVICE
    - CONFIG_CGROUPS
    - CONFIG_MEMCG

The kernel must support cgroups. The following requirements apply when running AWS IoT Greengrass with containers (p. 184):

- The memory cgroup must be enabled and mounted to allow AWS IoT Greengrass to set the memory limit for Lambda functions.
- The devices cgroup must be enabled and mounted if Lambda functions with local resource access (p. 200) are used to open files on the AWS IoT Greengrass core device.

- Others:
  - CONFIG_POSIX_MQUEUE
  - CONFIG_OVERLAY_FS
  - CONFIG_HAVE_ARCH_SECCOMP_FILTER
  - CONFIG_SECCOMP_FILTER
- CONFIG_KEYS
- CONFIG_SECCOMP
- CONFIG_SHMEM

- The root certificate for Amazon S3 and AWS IoT must be present in the system trust store.
- Libraries that support the AWS Lambda runtime required by the Lambda functions you want to run locally. Required libraries must be installed on the core and added to the PATH environment variable. Multiple libraries can be installed on the same core.
  - Python version 2.7 for functions that use the Python 2.7 runtime.
  - Python version 3.7 for functions that use the Python 3.7 runtime.
  - Node.js version 6.10 or later for functions that use the Node.js 6.10 runtime.
  - Node.js version 8.10 or later for functions that use the Node.js 8.10 runtime.
  - Java version 8 or later for functions that use the Java 8 runtime.

  **Note**
  Running Java on an OpenWrt distribution isn't officially supported. However, if your OpenWrt build has Java support, you might be able to run Lambda functions authored in Java on your OpenWrt devices.

- The following shell commands (not the BusyBox variants) are required by the over-the-air (OTA) update agent (p. 149):
  - wget
  - realpath
  - tar
  - readlink
  - basename
  - dirname
  - pidof
  - df
  - grep
  - umount
  - mv
  - gzip
  - mkdir
  - rm
  - ln
  - cut
  - cat

GGC v1.8

- Supported platforms:
  - Architecture: Armv7l; OS: Linux; Distribution: Raspbian Stretch, 2018-06-29. Other versions might work with AWS IoT Greengrass, but we recommend this because it is the officially supported distribution.
  - Architecture: x86_64; OS: Linux; Distribution: Amazon Linux (amzn-ami-hvm-2016.09.1.20170119-x86_64-ebs), Ubuntu 14.04 – 16.04
  - Architecture: Armv8 (AArch64); OS: Linux; Distribution: Arch Linux
  - Windows, macOS, and Linux platforms can run AWS IoT Greengrass in a Docker container. For more information, see the section called “Run AWS IoT Greengrass in a Docker Container” (p. 189).
- Linux platforms can run a version of AWS IoT Greengrass with limited functionality using the Greengrass snap, which is available through Snapcraft. For more information, see the section called “AWS IoT Greengrass Snap Software” (p. 19).

The following items are required:
- Minimum 128 MB RAM allocated to the AWS IoT Greengrass core device.
- Linux kernel version:
  - Linux kernel version 4.4 or later is required to support running AWS IoT Greengrass with containers (p. 181).
  - Linux kernel version 3.17 or later is required to support running AWS IoT Greengrass without containers. In this configuration, the default Lambda function containerization for the Greengrass group must be set to No container. For instructions, see the section called “Setting Default Containerization for Lambda Functions in a Group” (p. 184).
- GNU C Library (glibc) version 2.14 or later.
- The /var/run directory must be present on the device.
- The /dev/stdin, /dev/stdout, and /dev/stderr files must be available.
- Hardlink and softlink protection must be enabled on the device. Otherwise, AWS IoT Greengrass can only be run in insecure mode, using the –i flag.
- The following Linux kernel configurations must be enabled on the device:
  - Namespace:
    - CONFIG_IPC_NS
    - CONFIG_UTS_NS
    - CONFIG_USER_NS
    - CONFIG_PID_NS
  - Cgroups:
    - CONFIG_CGROUP_DEVICE
    - CONFIG_CGROUPS
    - CONFIG_MEMCG

The kernel must support cgroups. The following requirements apply when running AWS IoT Greengrass with containers (p. 184):
- The memory cgroup must be enabled and mounted to allow AWS IoT Greengrass to set the memory limit for Lambda functions.
- The devices cgroup must be enabled and mounted if Lambda functions with local resource access (p. 200) are used to open files on the AWS IoT Greengrass core device.
- Others:
  - CONFIG_POSIX_MQUEUE
  - CONFIG_OVERLAY_FS
  - CONFIG_HAVE_ARCH_SECCOMP_FILTER
  - CONFIG_SECCOMP_FILTER
  - CONFIG_KEYS
  - CONFIG_SECCOMP
  - CONFIG_SHMEM

- The root certificate for Amazon S3 and AWS IoT must be present in the system trust store.
- The following items are conditionally required:
  - Libraries that support the AWS Lambda runtime required by the Lambda functions you want to run locally. Required libraries must be installed on the core and added to the PATH environment variable. Multiple libraries can be installed on the same core.
  - Python version 2.7 for functions that use the Python 2.7 runtime.
- **Node.js** version 6.10 or later for functions that use the Node.js 6.10 runtime.
- **Java** version 8 or later for functions that use the Java 8 runtime.
- The following shell commands (not the BusyBox variants) are required by the **over-the-air (OTA) update agent** (p. 149):
  - `wget`
  - `realpath`
  - `tar`
  - `readlink`
  - `basename`
  - `dirname`
  - `pidof`
  - `df`
  - `grep`
  - `umount`
  - `mv`
  - `gzip`
  - `mkdir`
  - `rm`
  - `ln`
  - `cut`
  - `cat`

**GGC v1.7**

- **Supported platforms:**
  - Architecture: ARMv7l; OS: Linux; Distribution: Raspbian Stretch, 2018-06-29. While several versions may work with AWS IoT Greengrass, we recommend this as it is the officially supported distribution.
  - Architecture: x86_64; OS: Linux; Distribution: Amazon Linux (amzn-ami-hvm-2016.09.1.20170119-x86_64-ebs), Ubuntu 14.04 – 16.04
  - Architecture: ARMv8 (AArch64); OS: Linux; Distribution: Arch Linux
  - Windows, macOS, and Linux platforms can run AWS IoT Greengrass in a Docker container. For more information, see the section called “Run AWS IoT Greengrass in a Docker Container” (p. 189).

  - The following items are required:
    - Minimum 128 MB RAM allocated to the AWS IoT Greengrass core device.
    - Linux kernel version
      - Linux kernel version 4.4 or later is required to support running AWS IoT Greengrass with containers (p. 181).
      - Linux kernel version 3.17 or later is required to support running AWS IoT Greengrass without containers. In this configuration, the default Lambda function containerization for the Greengrass group must be set to **No container**. For instructions, see the section called “Setting Default Containerization for Lambda Functions in a Group” (p. 184).
    - **Glibc library** version 2.14 or greater.
    - The `/var/run` directory must be present on the device.
    - AWS IoT Greengrass requires hardlink and softlink protection to be enabled on the device. Without this, AWS IoT Greengrass can only be run in insecure mode, using the `-i` flag.
    - The `ggc_user` and `ggc_group` user and group must be present on the device.
• The following Linux kernel configurations must be enabled on the device:
  • Namespace:
    • CONFIG_IPC_NS
    • CONFIG_UTC_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
  • /dev/stdin, /dev/stdout, and /dev/stderr must be enabled.
  • The Linux kernel must support cgroups.
  • The memory cgroup must be enabled and mounted to allow AWS IoT Greengrass to set the memory limit for Lambda functions.
  • The root certificate for Amazon S3 and AWS IoT must be present in the system trust store.
  • The following items may be optional:
    • The devices cgroup must be enabled and mounted if Lambda functions with Local Resource Access (LRA) (p. 200) are used to open files on the AWS IoT Greengrass core device.
    • Python version 2.7 is required if Python Lambda functions are used. If so, ensure that it’s added to your PATH environment variable.
    • NodeJS version 6.10 or greater is required if Node.JS Lambda functions are used. If so, ensure that it’s added to your PATH environment variable.
    • Java version 8 or greater is required if Java Lambda functions are used. If so, ensure that it’s added to your PATH environment variable.
    • The following commands are required for Greengrass OTA Update Agent (p. 149): wget, realpath, tar, readlink, basename, dirname, pidof, df, grep, and umount.

For information about AWS IoT Greengrass limits, see AWS IoT Greengrass Limits in the Amazon Web Services General Reference.

AWS IoT Greengrass Downloads

You can use the following information to find and download software for use with AWS IoT Greengrass.

AWS IoT Greengrass Core Software

The AWS IoT Greengrass Core software extends AWS functionality onto an AWS IoT Greengrass core device, enabling local devices to act locally on the data they generate.
v1.9.4

New features in v1.9:

- Support for Python 3.7 and Node.js 8.10 Lambda runtimes. Lambda functions that use Python 3.7 and Node.js 8.10 runtimes can now run on an AWS IoT Greengrass core. (AWS IoT Greengrass continues to support the Python 2.7 and Node.js 6.10 runtimes.)
- Optimized MQTT connections. The Greengrass core establishes fewer connections with the AWS IoT Core. This change can reduce operational costs for charges that are based on the number of connections.
- Elliptic Curve (EC) key for the local MQTT server. The local MQTT server supports EC keys in addition to RSA keys. (The MQTT server certificate has an SHA-256 RSA signature, regardless of the key type.) For more information, see the section called “Security Principals” (p. 439).
- Support for OpenWrt. AWS IoT Greengrass Core software v1.9.2 or later can be installed on OpenWrt distributions with Armv8 (AArch64) and Armv7l architectures. Currently, OpenWrt does not support ML inference.
- Support for Armv6l. AWS IoT Greengrass Core software v1.9.3 or later can be installed on Raspbian distributions on Armv6l architectures (for example, on Raspberry Pi Zero devices).
- OTA updates on port 443 with ALPN. Greengrass cores that use port 443 for MQTT traffic now support over-the-air (OTA) software updates. AWS IoT Greengrass uses the Application Layer Protocol Network (ALPN) TLS extension to enable these connections. For more information, see OTA Updates of AWS IoT Greengrass Core Software (p. 148) and the section called “Connect on Port 443 or Through a Network Proxy” (p. 46).

To install the AWS IoT Greengrass Core software on your core device, download the package for your architecture, distribution, and operating system (OS), and then follow the steps in the Getting Started Guide (p. 64).

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Distribution</th>
<th>OS</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Armv8 (AArch64)</td>
<td>Ubuntu 18.04</td>
<td>Linux</td>
<td>Download</td>
</tr>
<tr>
<td>Armv8 (AArch64)</td>
<td>OpenWrt</td>
<td>Linux</td>
<td>Download</td>
</tr>
<tr>
<td>Armv7l</td>
<td>Raspbian</td>
<td>Linux</td>
<td>Download</td>
</tr>
<tr>
<td>Armv7l</td>
<td>OpenWrt</td>
<td>Linux</td>
<td>Download</td>
</tr>
<tr>
<td>Armv6l</td>
<td>Raspbian</td>
<td>Linux</td>
<td>Download</td>
</tr>
<tr>
<td>x86_64</td>
<td>Linux</td>
<td>Linux</td>
<td>Download</td>
</tr>
</tbody>
</table>

v1.8.4

- New features:
  - Configurable default access identity for Lambda functions in the group. This group-level setting determines the default permissions that are used to run Lambda functions. You can set the user ID, group ID, or both. Individual Lambda functions can override the default access identity of their group. For more information, see the section called “Setting the Default Access Identity for Lambda Functions in a Group” (p. 183).
  - HTTPS traffic over port 443. HTTPS communication can be configured to travel over port 443 instead of the default port 8443. This complements AWS IoT Greengrass support for the Application Layer Protocol Network (ALPN) TLS extension and allows all Greengrass messaging traffic—both MQTT and HTTPS—to use port 443. For more information, see the section called “Connect on Port 443 or Through a Network Proxy” (p. 46).
• Predictably named client IDs for AWS IoT connections. This change enables support for AWS IoT Device Defender and AWS IoT Lifecycle events, so you can receive notifications for connect, disconnect, subscribe, and unsubscribe events. Predictable naming also makes it easier to create logic around connection IDs (for example, to create subscribe policy templates based on certificate attributes). For more information, see the section called “Client IDs for MQTT Connections with AWS IoT” (p. 58).

Bug fixes and improvements:
• Fixed an issue with shadow synchronization and device certificate manager reconnection.
• General performance improvements and bug fixes.

To install the AWS IoT Greengrass Core software on your core device, download the package for your architecture, distribution, and operating system (OS), and then follow the steps in the Getting Started Guide (p. 64).

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Distribution</th>
<th>OS</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Armv8 (AArch64)</td>
<td>Ubuntu 14.04 - 16.04</td>
<td>Linux</td>
<td>Download</td>
</tr>
<tr>
<td>Armv7l</td>
<td>Raspbian</td>
<td>Linux</td>
<td>Download</td>
</tr>
<tr>
<td>x86_64</td>
<td>Linux</td>
<td>Linux</td>
<td>Download</td>
</tr>
</tbody>
</table>

v1.7.1

• New features:
  • Greengrass connectors provide built-in integration with local infrastructure, device protocols, AWS, and other cloud services. For more information, see Integrate with Services and Protocols Using Connectors (p. 283).
  • AWS IoT Greengrass extends AWS Secrets Manager to core devices, which makes your passwords, tokens, and other secrets available to connectors and Lambda functions. Secrets are encrypted in transit and at rest. For more information, see Deploy Secrets to the Core (p. 263).
  • Support for a hardware root of trust security option. For more information, see the section called “Hardware Security” (p. 448).
  • Isolation and permission settings that allow Lambda functions to run without Greengrass containers and to use the permissions of a specified user and group. For more information, see the section called “Controlling Greengrass Lambda Function Execution” (p. 178).
  • You can run AWS IoT Greengrass in a Docker container (on Windows, macOS, or Linux) by configuring your Greengrass group to run with no containerization. For more information, see the section called “Run AWS IoT Greengrass in a Docker Container” (p. 189).
  • MQTT messaging on port 443 with Application Layer Protocol Negotiation (ALPN) or connection through a network proxy. For more information, see the section called “Connect on Port 443 or Through a Network Proxy” (p. 46).
  • The Amazon SageMaker Neo deep learning runtime, which supports machine learning models that have been optimized by the Amazon SageMaker Neo deep learning compiler. For information about the Neo deep learning runtime, see the section called “Runtimes and Precompiled Framework Libraries for ML Inference” (p. 224).
  • Support for Raspbian Stretch (2018-06-27) on Raspberry Pi core devices.

Bug fixes and improvements:
• General performance improvements and bug fixes.

In addition, the following features are available with this release:
• The AWS IoT Device Tester for AWS IoT Greengrass, which you can use to verify that your CPU architecture, kernel configuration, and drivers work with AWS IoT Greengrass. For more information, see Using AWS IoT Device Tester for AWS IoT Greengrass (p. 482).

• The AWS IoT Greengrass Core software, AWS IoT Greengrass Core SDK, and AWS IoT Greengrass Machine Learning SDK packages are available for download through Amazon CloudFront. For more information, see the section called “AWS IoT Greengrass Downloads” (p. 16).

To install the AWS IoT Greengrass Core software on your core device, download the package for your architecture, distribution, and operating system (OS), and then follow the steps in the Getting Started Guide (p. 64).

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Distribution</th>
<th>OS</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Armv8 (AArch64)</td>
<td>Ubuntu 14.04 - 16.04</td>
<td>Linux</td>
<td>Download</td>
</tr>
<tr>
<td>Armv7l</td>
<td>Raspbian</td>
<td>Linux</td>
<td>Download</td>
</tr>
<tr>
<td>x86_64</td>
<td>Linux</td>
<td>Linux</td>
<td>Download</td>
</tr>
</tbody>
</table>

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

AWS IoT Greengrass Snap Software

Currently, AWS IoT Greengrass snap software is available for AWS IoT Greengrass core version 1.8 only.

The AWS IoT Greengrass snap software download makes it possible for you to run a version of AWS IoT Greengrass with limited functionality on Linux cloud, desktop, and IoT environments through convenient containerized software packages. These packages, or snaps, contain the AWS IoT Greengrass software and its dependencies. You can download and use these packages on your Linux environments as-is.

The AWS IoT Greengrass snap is updated automatically with new versions of the AWS IoT Greengrass core software and associated dependencies. You can turn off these automatic updates at any time.

Note
The AWS IoT Greengrass snap allows you to run a version of AWS IoT Greengrass with limited functionality on your Linux environments. Currently, Java, Node.js, and native Lambda functions are not supported. Machine learning inference, connectors, and noncontainerized Lambda functions are also not supported.

Getting Started with AWS IoT Greengrass Snap

Because the prepackaged AWS IoT Greengrass snap is designed to use system defaults, you might need to perform these other steps:

• The AWS IoT Greengrass snap is configured to use default Greengrass user and group configurations. This allows it to work easily with Greengrass groups or Lambda functions that run as root. If you need to use Greengrass groups or Lambda functions that do not run as root, update these configurations and add them to your system.

• The AWS IoT Greengrass snap uses many interfaces that must be connected before it can operate normally. These interfaces are connected automatically during setup. If you use other options while you set up your snap, you might need to connect these interfaces manually.

For more information about the AWS IoT Greengrass snap and these modifications, see Greengrass Snap Release Notes.
1. Install and upgrade snapd by running the following command in your device's terminal:

```
sudo apt-get update && sudo apt-get upgrade snapd
```

2. If you need to use Greengrass groups or Lambda functions that do not run as root, update your default Greengrass user and group configurations, and add them to your system. For more information about updating user and group configurations with AWS IoT Greengrass, see ??? (p. 183).

   • For the Ubuntu Core system:
     • To add the `ggc_user` user, use:

       ```
sudo adduser --extrausers --system ggc_user
```

     • To add the `ggc_group` group, use:

       ```
sudo addgroup --extrausers --system ggc_group
```

   • For the Ubuntu classic system:
     • To add the `ggc_user` user to an Ubuntu classic system, omit the --extrausers flag and use:

       ```
sudo adduser --system ggc_user
```

     • To add the `ggc_group` to an Ubuntu classic system, omit the --extrausers flag and use:

       ```
sudo addgroup --system ggc_group
```

3. In your terminal, run the following command to install the Greengrass snap:

```
sudo snap install aws-iot-greengrass
```

**Note**  
You can also use the AWS IoT Greengrass snap download link to install the Greengrass snap locally. If you are installing locally from this file and do not have the associated assertions, use the --dangerous flag:

```
sudo snap install --dangerous aws-iot-greengrass*.snap
```

The --dangerous flag interferes with the AWS IoT Greengrass snap's ability to connect its required interfaces. If you use this flag, you need to manually connect the required interfaces using the `snap connect` command. For more information, see Greengrass Snap Release Notes.

4. After the snap is installed, run the following command to add your Greengrass certificate and configuration files:

```
sudo snap set aws-iot-greengrass gg-certs=/path-to-the-certs/22e592db.tgz
```

**Note**  
If necessary, you can troubleshoot issues by viewing the AWS IoT Greengrass core logs, particularly `runtime.log`. You can print the contents of `runtime.log` to your terminal by running the following command:

```
sudo cat /var/snap/aws-iot-greengrass/current/ggc-writable/var/log/system/runtime.log
```
5. Run the following command to validate that your setup is functioning correctly:

```
$ snap services aws-iot-greengrass
```

You should see the following response:

<table>
<thead>
<tr>
<th>Service</th>
<th>Startup</th>
<th>Current</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>aws-iot-greengrass.greengrassd</td>
<td>enabled</td>
<td>active</td>
<td>-</td>
</tr>
</tbody>
</table>

Your Greengrass setup is now complete. You can now use the AWS IoT Greengrass console, AWS REST API, or AWS CLI to deploy the Greengrass groups associated with this snap. For information about using the console to deploy a Greengrass group, see the [Deploy Cloud Configurations to an AWS IoT Greengrass Core Device](#). For information about using the CLI or REST API to deploy a Greengrass group, see [CreateDeployment](#) in the AWS IoT Greengrass API Reference.

For more information about configuring local resource access with snap AppArmor confinement, using the snapd REST API, and configuring snap interfaces, see [Greengrass Snap Release Notes](#).

## AWS IoT Greengrass Docker Software

AWS provides a Dockerfile and Docker image that make it easier for you to run AWS IoT Greengrass in a Docker container.

### Dockerfile

Source code for building custom AWS IoT Greengrass container images. The image can be modified to run on different platform architectures or to reduce the image size. For instructions, see the README file.

Choose the AWS IoT Greengrass Core software version.

- v1.9.4
  - Docker v1.9.4.
- v1.8.1
  - Docker v1.8.1.
- v1.7.1
  - Docker v1.7.1.

### Docker image

Docker image with the AWS IoT Greengrass Core software and dependencies installed. Prebuilt images can help you get started quickly and experiment with AWS IoT Greengrass.

- Docker image from [Docker Hub](#).
- Docker image from Amazon Elastic Container Registry (Amazon ECR). For more information, see the section called “Run AWS IoT Greengrass in a Docker Container” (p. 189).

By downloading this software you agree to the [Greengrass Core Software License Agreement](#).
AWS IoT Greengrass Core SDK Software

Lambda functions use the AWS IoT Greengrass Core SDK to interact with the AWS IoT Greengrass core locally. This allows deployed Lambda functions to:

- Exchange MQTT messages with AWS IoT.
- Exchange MQTT messages with connectors, devices, and other Lambda functions in the Greengrass group.
- Interact with the local shadow service.
- Invoke other local Lambda functions.
- Access secret resources (p. 263).

Download the AWS IoT Greengrass Core SDK for your language or platform from GitHub.

- AWS IoT Greengrass Core SDK for Java
- AWS IoT Greengrass Core SDK for Node.js
- AWS IoT Greengrass Core SDK for Python
- AWS IoT Greengrass Core SDK for C

If you're running Python Lambda functions, you can also use pip to install the AWS IoT Greengrass Core SDK for Python on the core device. Then you can deploy your functions without including the SDK in the Lambda function deployment package. For more information, see greengrassdk.

**Note**
To use pip to install the Python SDK, run the following command in your core device terminal.

```
pip install greengrasssdk
```

AWS IoT Greengrass Machine Learning Runtimes and Precompiled Libraries

Machine learning runtimes and libraries are required for your ML models to perform inference on Greengrass devices.

Download the model type for your platform.

Raspberry Pi

Choose the download link for your model type.

By downloading this software you agree to the associated license.

<table>
<thead>
<tr>
<th>Model type</th>
<th>Version</th>
<th>License</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>MXNet</td>
<td>1.2.1</td>
<td>Apache License 2.0</td>
<td>Download</td>
</tr>
<tr>
<td>TensorFlow</td>
<td>1.4.0</td>
<td>Apache License 2.0</td>
<td>Download</td>
</tr>
<tr>
<td>Deep Learning Runtime</td>
<td>1.0.0</td>
<td>Greengrass License</td>
<td>Download</td>
</tr>
<tr>
<td>Chainer</td>
<td>4.0.0</td>
<td>MIT License</td>
<td>Download</td>
</tr>
</tbody>
</table>
Nvidia Jetson TX2

Choose the download link for your model type.

By downloading this software you agree to the associated license.

<table>
<thead>
<tr>
<th>Model type</th>
<th>Version</th>
<th>License</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>MXNet</td>
<td>1.2.1</td>
<td>Apache License 2.0</td>
<td>Download</td>
</tr>
<tr>
<td>TensorFlow</td>
<td>1.10.0</td>
<td>Apache License 2.0</td>
<td>Download</td>
</tr>
<tr>
<td>Deep Learning Runtime</td>
<td>1.0.0</td>
<td>Greengrass License</td>
<td>Download</td>
</tr>
<tr>
<td>Chainer</td>
<td>4.0.0</td>
<td>MIT License</td>
<td>Download</td>
</tr>
</tbody>
</table>

Intel Atom

Choose the download link for your model type.

By downloading this software you agree to the associated license.

<table>
<thead>
<tr>
<th>Model type</th>
<th>Version</th>
<th>License</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>MXNet</td>
<td>1.2.1</td>
<td>Apache License 2.0</td>
<td>Download</td>
</tr>
<tr>
<td>TensorFlow</td>
<td>1.4.0</td>
<td>Apache License 2.0</td>
<td>Download</td>
</tr>
<tr>
<td>Deep Learning Runtime</td>
<td>1.0.0</td>
<td>Greengrass License</td>
<td>Download</td>
</tr>
<tr>
<td>Chainer</td>
<td>4.0.0</td>
<td>MIT License</td>
<td>Download</td>
</tr>
</tbody>
</table>

AWS IoT Greengrass ML SDK Software

The AWS IoT Greengrass Machine Learning SDK (p. 176) enables the Lambda functions you author to consume a local machine learning model and send data to the ML Feedback (p. 315) connector for uploading and publishing.

v1.1.0

- Python 3.7 or 2.7 - Current version.

v1.0.0

- Python 2.7.

We Want to Hear from You

We welcome your feedback. To contact us, visit the AWS IoT Greengrass Forum.
Configure the AWS IoT Greengrass Core

An AWS IoT Greengrass core is an AWS IoT thing (device). Like other AWS IoT devices, a core exists in the registry, has a device shadow, and uses a device certificate to authenticate with AWS IoT. The core device runs the AWS IoT Greengrass Core software, which enables it to manage local processes for Greengrass groups, such as communication, shadow sync, and token exchange.

The AWS IoT Greengrass Core software provides the following functionality:

- Deployment and local execution of connectors and Lambda functions.
- Secure, encrypted storage of local secrets and controlled access by connectors and Lambda functions.
- MQTT messaging over the local network between devices, connectors, and Lambda functions using managed subscriptions.
- MQTT messaging between AWS IoT and devices, connectors, and Lambda functions using managed subscriptions.
- Secure connections between devices and the cloud using device authentication and authorization.
- Local shadow synchronization of devices. Shadows can be configured to sync with the cloud.
- Controlled access to local device and volume resources.
- Deployment of cloud-trained machine learning models for running local inference.
- Automatic IP address detection that enables devices to discover the Greengrass core device.
- Central deployment of new or updated group configuration. After the configuration data is downloaded, the core device is restarted automatically.
- Secure, over-the-air software updates of user-defined Lambda functions.

AWS IoT Greengrass Core Configuration File

The configuration file for the AWS IoT Greengrass Core software is `config.json`. It is located in the `/greengrass-root/config` directory.

**Note**

`greengrass-root` represents the path where the AWS IoT Greengrass Core software is installed on your device. If you installed the software by following the steps in the Getting Started (p. 64) tutorial, then this is the `/greengrass` directory.

If you use the Easy group creation option from the AWS IoT Greengrass console, then the `config.json` file is deployed to the core device in a working state.

You can review the contents of this file by running the following command:

```
cat /greengrass-root/config/config.json
```

The following is an example `config.json` file. This is the version that’s generated when you create the core from the AWS IoT Greengrass console.

GGC v1.9

```
{
  "coreThing" : {
    "caPath" : "root.ca.pem",
    "certPath" : "hash.cert.pem",
    "keyPath" : "hash.private.key",
    "iotHost" : "host-prefix-ats.iot.region.amazonaws.com",
    "ggHost" : "greengrass-ats.iot.region.amazonaws.com",
    "keepAlive" : 600
  },
```
"runtime": {
  "cgroup": {
    "useSystemd": "yes"
  }
},
"managedRespawn": false,
"crypto": {
  "principals": {
    "SecretsManager": {
      "privateKeyPath": "file:///greengrass/certs/hash.private.key"
    },
    "IoTCertificate": {
      "privateKeyPath": "file:///greengrass/certs/hash.private.key",
      "certificatePath": "file:///greengrass/certs/hash.cert.pem"
    }
  },
  "caPath": "file:///greengrass/certs/root.ca.pem"
}
}

The `config.json` file supports the following properties:

**coreThing**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>caPath</td>
<td>The path to the AWS IoT root CA relative to the <code>./greengrass-root/certs</code> directory.</td>
<td>For backward compatibility with versions earlier than 1.7.0. This property is ignored when the <code>crypto</code> object is present. <strong>Note</strong> Make sure that your endpoints correspond to your certificate type (p. 45).</td>
</tr>
<tr>
<td>certPath</td>
<td>The path to the core device certificate relative to the <code>./greengrass-root/certs</code> directory.</td>
<td>For backward compatibility with versions earlier than 1.7.0. This property is ignored when the <code>crypto</code> object is present.</td>
</tr>
<tr>
<td>keyPath</td>
<td>The path to the core private key relative to <code>./greengrass-root/certs</code> directory.</td>
<td>For backward compatibility with versions earlier than 1.7.0. This property is ignored when the <code>crypto</code> object is present.</td>
</tr>
<tr>
<td>thingArn</td>
<td>The Amazon Resource Name (ARN) of the AWS IoT thing that represents the AWS IoT Greengrass core device.</td>
<td>Find this for your core in the AWS IoT Greengrass console under <strong>Cores</strong>, or by running the <code>aws greengrass get-core-definition-version</code> CLI command.</td>
</tr>
<tr>
<td>iotHost</td>
<td>Your AWS IoT endpoint.</td>
<td>Find this in the AWS IoT console under <strong>Settings</strong>, or by running the <code>aws iot describe-endpoint --endpoint-type iot:Data-ATS</code> CLI command.</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>This command returns the Amazon Trust Services (ATS) endpoint. For more information, see the Server Authentication documentation.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Notes</td>
<td>Make sure that your endpoints correspond to your certificate type (p. 45).</td>
<td>Make sure that your endpoints correspond to your region.</td>
</tr>
<tr>
<td>ggHost</td>
<td>Your AWS IoT Greengrass endpoint.</td>
<td>This is your iotHost endpoint with the host prefix replaced by greengrass (for example, greengrass-ats.iot.region.amazonaws.com). Use the same AWS Region as iotHost.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Note</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Make sure that your endpoints correspond to your certificate type (p. 45).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Make sure that your endpoints correspond to your region.</td>
</tr>
<tr>
<td>iotMqttPort</td>
<td>Optional. The port number to use for MQTT communication with AWS IoT.</td>
<td>Valid values are 8883 or 443. The default value is 8883. For more information, see Connect on Port 443 or Through a Network Proxy (p. 46).</td>
</tr>
<tr>
<td>iotHttpPort</td>
<td>Optional. The port number used to create HTTPS connections to AWS IoT.</td>
<td>Valid values are 8443 or 443. The default value is 8443. For more information, see Connect on Port 443 or Through a Network Proxy (p. 46).</td>
</tr>
<tr>
<td>ggHttpPort</td>
<td>Optional. The port number used to create HTTPS connections to the AWS IoT Greengrass service.</td>
<td>Valid values are 8443 or 443. The default value is 8443. For more information, see Connect on Port 443 or Through a Network Proxy (p. 46).</td>
</tr>
<tr>
<td>keepAlive</td>
<td>Optional. The MQTT KeepAlive period, in seconds.</td>
<td>Valid range is between 30 and 1200 seconds. The default value is 600.</td>
</tr>
</tbody>
</table>
### AWS IoT Greengrass Core Configuration File

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>networkProxy</td>
<td>Optional. An object that defines a proxy server to connect to.</td>
<td>This can be an HTTP or HTTPS proxy. For more information, see Connect on Port 443 or Through a Network Proxy (p. 46).</td>
</tr>
</tbody>
</table>

#### runtime

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>postStartHealthCheckTimeout</td>
<td>Optional. The time (in milliseconds) that the Greengrass daemon waits after starting for the health check to finish.</td>
<td>The default timeout is 30 seconds (30000 ms).</td>
</tr>
</tbody>
</table>

#### cgroup

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>useSystemd</td>
<td>Indicates whether your device uses systemd.</td>
<td>Valid values are yes or no. Run the check_ggc_dependencies script in Module 1 (p. 65) to see if your device uses systemd.</td>
</tr>
</tbody>
</table>

#### crypto

The crypto object is added in v1.7.0. It introduces properties that support private key storage on a hardware security module (HSM) through PKCS#11 and local secret storage. For more information, see the section called “Security Principals” (p. 439), the section called “Hardware Security” (p. 448), and Deploy Secrets to the Core (p. 263). Configurations for private key storage on HSMs or in the file system are supported.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>caPath</td>
<td>The absolute path to the AWS IoT root CA.</td>
<td>Must be a file URI of the form: file:///absolute/path/to/file.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note</strong> Make sure that your endpoints correspond to your certificate type (p. 45).</td>
</tr>
</tbody>
</table>

#### PKCS11

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenSSLEngine</td>
<td>Optional. The absolute path to the OpenSSL engine .so file to enable PKCS#11 support on OpenSSL.</td>
<td>Must be a path to a file on the file system. This property is required if you’re using the Greengrass OTA update agent with hardware security. For more information, see the section</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>---------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>P11Provider</strong></td>
<td>The absolute path to the PKCS#11 implementation's libdl-loadable library.</td>
<td>Must be a path to a file on the file system.</td>
</tr>
<tr>
<td><strong>slotLabel</strong></td>
<td>The slot label that's used to identify the hardware module.</td>
<td>Must conform to PKCS#11 label specifications.</td>
</tr>
<tr>
<td><strong>slotUserPin</strong></td>
<td>The user pin that's used to authenticate the Greengrass core to the module.</td>
<td>Must have sufficient permissions to perform C_Sign with the configured private keys.</td>
</tr>
<tr>
<td><strong>IoTCertificate</strong></td>
<td>The certificate and private key that the core uses to make requests to AWS IoT.</td>
<td></td>
</tr>
<tr>
<td><strong>IoTCertificate</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.privateKeyPath</td>
<td>The path to the core private key.</td>
<td>For file system storage, must be a file URI of the form: file://absolute/path/to/file.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For HSM storage, must be an RFC 7512 PKCS#11 path that specifies the object label.</td>
</tr>
<tr>
<td>.certificatePath</td>
<td>The absolute path to the core device certificate.</td>
<td>Must be a file URI of the form: file://absolute/path/to/file.</td>
</tr>
<tr>
<td><strong>MQTTServerCertificate</strong></td>
<td>Optional. The private key that the core uses in combination with the certificate to act as an MQTT server or gateway.</td>
<td></td>
</tr>
<tr>
<td>.privateKeyPath</td>
<td>The path to the local MQTT server private key.</td>
<td>Use this value to specify your own private key for the local MQTT server.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For file system storage, must be a file URI of the form: file://absolute/path/to/file.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For HSM storage, must be an RFC 7512 PKCS#11 path that specifies the object label.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If this property is omitted, AWS IoT Greengrass rotates the key based your rotation settings. If specified, the customer is responsible for rotating the key.</td>
</tr>
<tr>
<td><strong>SecretsManager</strong></td>
<td>The private key that secures the data key used for encryption.</td>
<td>For more information, see Deploy Secrets to the Core (p. 263).</td>
</tr>
</tbody>
</table>

**principals**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IoTCertificate</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.privateKeyPath</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.certificatePath</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>MQTTServerCertificate</strong></td>
<td>Optional. The private key that the core uses in combination with the certificate to act as an MQTT server or gateway.</td>
<td></td>
</tr>
<tr>
<td>.privateKeyPath</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SecretsManager**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
<td>-------</td>
</tr>
<tr>
<td>SecretsManager.privateKeyPath</td>
<td>The path to the local secrets manager private key.</td>
<td>Only an RSA key is supported. For file system storage, must be a file URI of the form: file:///absolute/path/to/file. For HSM storage, must be an RFC 7512 PKCS#11 path that specifies the object label. The private key must be generated using the PKCS#1 v1.5 padding mechanism.</td>
</tr>
</tbody>
</table>

The following configuration properties are also supported:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>mqttMaxConnectionRetryInterval</td>
<td>Optional. The maximum interval (in seconds) between MQTT connection retries if the connection is dropped.</td>
<td>Specify this value as an unsigned integer. The default is 60.</td>
</tr>
<tr>
<td>managedRespawn</td>
<td>Optional. Indicates that the OTA agent needs to run custom code before an update.</td>
<td>Valid values are true or false. For more information, see OTA Updates of AWS IoT Greengrass Core Software (p. 148).</td>
</tr>
<tr>
<td>writeDirectory</td>
<td>Optional. The write directory where AWS IoT Greengrass creates all read-write resources.</td>
<td>For more information, see Configure a Write Directory for AWS IoT Greengrass (p. 52).</td>
</tr>
</tbody>
</table>

GGC v1.8

```json
{
  "coreThing" : {
    "caPath" : "root.ca.pem",
    "certPath" : "hash.cert.pem",
    "keyPath" : "hash.private.key",
    "iotHost" : "host-prefix-ats.iot.region.amazonaws.com",
    "ggHost" : "greengrass-ats.iot.region.amazonaws.com",
    "keepAlive" : 600
  },
  "runtime" : {
    "cgroup" : {
      "useSystemd" : "yes"
    },
  },
  "managedRespawn" : false,
  "crypto" : {
    "principals" : {
      "SecretsManager" : {
        "privateKeyPath" : "file:///greengrass/certs/hash.private.key"
      }
    }
  }
}
```
The `config.json` file supports the following properties:

## coreThing

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>caPath</td>
<td>The path to the AWS IoT root CA relative to the <code>/greengrass-root/certs</code> directory.</td>
<td>For backward compatibility with versions earlier than 1.7.0. This property is ignored when the crypto object is present.</td>
</tr>
<tr>
<td>certPath</td>
<td>The path to the core device certificate relative to the <code>/greengrass-root/certs</code> directory.</td>
<td>For backward compatibility with versions earlier than 1.7.0. This property is ignored when the crypto object is present.</td>
</tr>
<tr>
<td>keyPath</td>
<td>The path to the core private key relative to <code>/greengrass-root/certs</code> directory.</td>
<td>For backward compatibility with versions earlier than 1.7.0. This property is ignored when the crypto object is present.</td>
</tr>
<tr>
<td>thingArn</td>
<td>The Amazon Resource Name (ARN) of the AWS IoT thing that represents the AWS IoT Greengrass core device.</td>
<td>Find this for your core in the AWS IoT Greengrass console under Cores, or by running the <code>aws greengrass get-core-definition-version</code> CLI command.</td>
</tr>
<tr>
<td>iotHost</td>
<td>Your AWS IoT endpoint.</td>
<td>Find this in the AWS IoT console under Settings, or by running the <code>aws iot describe-endpoint --endpoint-type iot:Data-ATS</code> CLI command.</td>
</tr>
</tbody>
</table>

Note: Make sure that your endpoints correspond to your certificate type (p. 45).
## AWS IoT Greengrass Core Configuration File

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ggHost</td>
<td>Your AWS IoT Greengrass endpoint.</td>
<td>This is your <code>iotHost</code> endpoint with the host prefix replaced by <code>greengrass</code> (for example, <code>greengrass-ats.iot.region.amazonaws.com</code>). Use the same AWS Region as <code>iotHost</code>. Note: Make sure that your endpoints correspond to your certificate type (p. 45). Make sure your endpoints correspond to your AWS Region.</td>
</tr>
<tr>
<td>iotMqttPort</td>
<td>Optional. The port number to use for MQTT communication with AWS IoT.</td>
<td>Valid values are 8883 or 443. The default value is 8883. For more information, see Connect on Port 443 or Through a Network Proxy (p. 46).</td>
</tr>
<tr>
<td>iotHttpPort</td>
<td>Optional. The port number used to create HTTPS connections to AWS IoT.</td>
<td>Valid values are 8443 or 443. The default value is 8443. For more information, see Connect on Port 443 or Through a Network Proxy (p. 46).</td>
</tr>
<tr>
<td>ggHttpPort</td>
<td>Optional. The port number used to create HTTPS connections to the AWS IoT Greengrass service.</td>
<td>Valid values are 8443 or 443. The default value is 8443. For more information, see Connect on Port 443 or Through a Network Proxy (p. 46).</td>
</tr>
<tr>
<td>keepAlive</td>
<td>Optional. The MQTT KeepAlive period, in seconds.</td>
<td>Valid range is between 30 and 1200 seconds. The default value is 600.</td>
</tr>
<tr>
<td>networkProxy</td>
<td>Optional. An object that defines a proxy server to connect to.</td>
<td>This can be an HTTP or HTTPS proxy. For more information, see Connect on Port 443 or Through a Network Proxy (p. 46).</td>
</tr>
</tbody>
</table>

### runtime

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>cgroup</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Field | Description | Notes
--- | --- | ---
**useSystemd** | Indicates whether your device uses systemd. | Valid values are yes or no. Run the check_ggc_dependencies script in Module 1 (p. 65) to see if your device uses systemd.

**crypto**
The crypto object is added in v1.7.0. It introduces properties that support private key storage on a hardware security module (HSM) through PKCS#11 and local secret storage. For more information, see the section called “Security Principals” (p. 439), the section called “Hardware Security” (p. 448), and Deploy Secrets to the Core (p. 263). Configurations for private key storage on HSMs or in the file system are supported.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>caPath</td>
<td>The absolute path to the AWS IoT root CA.</td>
<td>Must be a file URI of the form: file:///absolute/path/to/file. <strong>Note</strong> Make sure that your endpoints correspond to your certificate type (p. 45).</td>
</tr>
</tbody>
</table>

**PKCS11**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenSSLEngine</td>
<td>Optional. The absolute path to the OpenSSL engine .so file to enable PKCS#11 support on OpenSSL.</td>
<td>Must be a path to a file on the file system. This property is required if you're using the Greengrass OTA update agent with hardware security. For more information, see the section called “Configure OTA Updates” (p. 454).</td>
</tr>
<tr>
<td>P11Provider</td>
<td>The absolute path to the PKCS#11 implementation's libdl-loadable library.</td>
<td>Must be a path to a file on the file system.</td>
</tr>
<tr>
<td>slotLabel</td>
<td>The slot label that's used to identify the hardware module.</td>
<td>Must conform to PKCS#11 label specifications.</td>
</tr>
<tr>
<td>slotUserPin</td>
<td>The user pin that's used to authenticate the Greengrass core to the module.</td>
<td>Must have sufficient permissions to perform C_Sign with the configured private keys.</td>
</tr>
</tbody>
</table>

**principals**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IoTCertificate</td>
<td>The certificate and private key that the core uses to make requests to AWS IoT.</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>IoTCertificate.privateKeyPath</td>
<td>The path to the core private key.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>IoTCertificate.certificatePath</td>
<td>The absolute path to the core device certificate.</td>
</tr>
<tr>
<td>MQTTServerCertificate</td>
<td>Optional. The private key that the core uses in combination with the certificate to act as an MQTT server or gateway.</td>
</tr>
<tr>
<td>MQTTServerCertificate.privateKeyPath</td>
<td>The path to the local MQTT server private key.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>SecretsManager</td>
<td>The private key that secures the data key used for encryption.</td>
</tr>
<tr>
<td>SecretsManager.privateKeyPath</td>
<td>The path to the local secrets manager private key.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following configuration properties are also supported:
### AWS IoT Greengrass Developer Guide
### AWS IoT Greengrass Core Configuration File

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>mqttMaxConnectionRetryInterval</td>
<td>Optional. The maximum interval (in seconds) between MQTT connection retries if the connection is dropped.</td>
<td>Specify this value as an unsigned integer. The default is 60.</td>
</tr>
<tr>
<td>managedRespawn</td>
<td>Optional. Indicates that the OTA agent needs to run custom code before an update.</td>
<td>Valid values are true or false. For more information, see OTA Updates of AWS IoT Greengrass Core Software (p. 148).</td>
</tr>
<tr>
<td>writeDirectory</td>
<td>Optional. The write directory where AWS IoT Greengrass creates all read-write resources.</td>
<td>For more information, see Configure a Write Directory for AWS IoT Greengrass (p. 52).</td>
</tr>
</tbody>
</table>

GGC v1.7

```json
{
    "coreThing": {
        "caPath": "root.ca.pem",
        "certPath": "hash.cert.pem",
        "keyPath": "hash.private.key",
        "thingArn": "arn:aws:iot:region:account-id:thing/core-thing-name",
        "iotHost": "host-prefix-ats.iot.region.amazonaws.com",
        "ggHost": "greengrass-ats.iot.region.amazonaws.com",
        "keepAlive": 600
    },
    "runtime": {
        "cgroup": {
            "useSystemd": "yes"
        }
    },
    "managedRespawn": false,
    "crypto": {
        "principals": {
            "SecretsManager": {
                "privateKeyPath": "file:///greengrass/certs/hash.private.key"
            },
            "IoTCertificate": {
                "privateKeyPath": "file:///greengrass/certs/hash.private.key",
                "certificatePath": "file:///greengrass/certs/hash.cert.pem"
            }
        },
        "caPath": "file:///greengrass/certs/root.ca.pem"
    }
}
```

The **config.json** file supports the following properties:

- **coreThing**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>caPath</td>
<td>The path to the AWS IoT root CA relative to the</td>
<td>For backward compatibility with versions earlier than 1.7.0.</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>/greengrass-root/certs directory.</td>
<td>This property is ignored when the crypto object is present.</td>
<td><strong>Note</strong> Make sure that your endpoints correspond to your certificate type (p. 45).</td>
</tr>
<tr>
<td>certPath</td>
<td>The path to the core device certificate relative to the /greengrass-root/certs directory.</td>
<td>For backward compatibility with versions earlier than 1.7.0. This property is ignored when the crypto object is present.</td>
</tr>
<tr>
<td>keyPath</td>
<td>The path to the core private key relative to /greengrass-root/certs directory.</td>
<td>For backward compatibility with versions earlier than 1.7.0. This property is ignored when the crypto object is present.</td>
</tr>
<tr>
<td>thingArn</td>
<td>The Amazon Resource Name (ARN) of the AWS IoT thing that represents the AWS IoT Greengrass core device.</td>
<td>Find this for your core in the AWS IoT Greengrass console under Cores, or by running the aws greengrass get-core-definition-version CLI command.</td>
</tr>
</tbody>
</table>
| iotHost | Your AWS IoT endpoint.                                                       | Find this in the AWS IoT console under Settings, or by running the aws iot describe-endpoint --endpoint-type iot:Data-ATS CLI command.  
This command returns the Amazon Trust Services (ATS) endpoint. For more information, see the Server Authentication documentation.  
**Note** Make sure that your endpoints correspond to your certificate type (p. 45). Make sure your endpoints correspond to your AWS Region. |
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ggHost</td>
<td>Your AWS IoT Greengrass endpoint.</td>
<td>This is your iotHost endpoint with the host prefix replaced by greengrass (for example, greengrass-ats.iot.region.amazonaws.com). Use the same AWS Region as iotHost.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note</strong> Make sure that your endpoints correspond to your certificate type (p. 45). Make sure your endpoints correspond to your AWS Region.</td>
</tr>
<tr>
<td>iotMqttPort</td>
<td>Optional. The port number to use for MQTT communication with AWS IoT.</td>
<td>Valid values are 8883 or 443. The default value is 8883. For more information, see Connect on Port 443 or Through a Network Proxy (p. 46).</td>
</tr>
<tr>
<td>keepAlive</td>
<td>Optional. The MQTT KeepAlive period, in seconds.</td>
<td>Valid range is between 30 and 1200 seconds. The default value is 600.</td>
</tr>
<tr>
<td>networkProxy</td>
<td>Optional. An object that defines a proxy server to connect to.</td>
<td>This can be an HTTP or HTTPS proxy. For more information, see Connect on Port 443 or Through a Network Proxy (p. 46).</td>
</tr>
</tbody>
</table>

### runtime

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>cgroup</td>
<td></td>
<td></td>
</tr>
<tr>
<td>useSystemd</td>
<td>Indicates whether your device uses systemd.</td>
<td>Valid values are yes or no. Run the check_ggc_dependencies script in Module 1 (p. 65) to see if your device uses systemd.</td>
</tr>
</tbody>
</table>

### crypto

The crypto object, added in v1.7.0, introduces properties that support private key storage on a hardware security module (HSM) through PKCS#11 and local secret storage. For more information, see the section called "Hardware Security" (p. 448) and Deploy Secrets to the Core (p. 263). Configurations for private key storage on HSMs or in the file system are supported.
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>caPath</td>
<td>The absolute path to the AWS IoT root CA.</td>
<td>Must be a file URI of the form: file:///absolute/path/to/file. <strong>Note</strong>: Make sure that your endpoints correspond to your certificate type (p. 45).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PKCS11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenSSLEngine</td>
<td>Optional. The absolute path to the OpenSSL engine .so file to enable PKCS#11 support on OpenSSL.</td>
<td>Must be a path to a file on the file system.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This property is required if you’re using the Greengrass OTA update agent with hardware security. For more information, see the section called “Configure OTA Updates” (p. 454).</td>
</tr>
<tr>
<td>P11Provider</td>
<td>The absolute path to the PKCS#11 implementation’s libdl-loadable library.</td>
<td>Must be a path to a file on the file system.</td>
</tr>
<tr>
<td>slotLabel</td>
<td>The slot label that’s used to identify the hardware module.</td>
<td>Must conform to PKCS#11 label specifications.</td>
</tr>
<tr>
<td>slotUserPin</td>
<td>The user pin that’s used to authenticate the Greengrass core to the module.</td>
<td>Must have sufficient permissions to perform C_Sign with the configured private keys.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>principals</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IoTCertificate</td>
<td>The certificate and private key that the core uses to make requests to AWS IoT.</td>
<td></td>
</tr>
<tr>
<td>IoTCertificate.privateKeyPath</td>
<td>The path to the core private key.</td>
<td>For file system storage, must be a file URI of the form: file:///absolute/path/to/file. For HSM storage, must be an RFC 7512 PKCS#11 path that specifies the object label.</td>
</tr>
<tr>
<td>IoTCertificate.certificatePath</td>
<td>The absolute path to the core device certificate.</td>
<td>Must be a file URI of the form: file:///absolute/path/to/file.</td>
</tr>
<tr>
<td>MQTTServerCertificate</td>
<td>Optional. The private key that the core uses in combination with the certificate to act as an MQTT server or gateway.</td>
<td></td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
<td>-------</td>
</tr>
<tr>
<td><strong><code>MQTTServerCertificate.privateKeyPath</code></strong></td>
<td>The path to the local MQTT server private key.</td>
<td>Use this value to specify your own private key for the local MQTT server. For file system storage, must be a file URI of the form: file:///absolute/path/to/file. For HSM storage, must be an RFC 7512 PKCS#11 path that specifies the object label. If this property is omitted, AWS IoT Greengrass rotates the key based your rotation settings. If specified, the customer is responsible for rotating the key.</td>
</tr>
<tr>
<td><strong>SecretsManager</strong></td>
<td>The private key that secures the data key used for encryption. For more information, see <em>Deploy Secrets to the Core</em> (p. 263).</td>
<td></td>
</tr>
<tr>
<td><strong>SecretsManager.privateKeyPath</strong></td>
<td>The path to the local secrets manager private key. Only an RSA key is supported. For file system storage, must be a file URI of the form: file:///absolute/path/to/file. For HSM storage, must be an RFC 7512 PKCS#11 path that specifies the object label. The private key must be generated using the PKCS#1 v1.5 padding mechanism.</td>
<td></td>
</tr>
</tbody>
</table>

The following configuration properties are also supported:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mqttMaxConnectionRetryInterval</code></td>
<td>Optional. The maximum interval (in seconds) between MQTT connection retries if the connection is dropped.</td>
<td>Specify this value as an unsigned integer. The default is 60.</td>
</tr>
<tr>
<td><code>managedRespawn</code></td>
<td>Optional. Indicates that the OTA agent needs to run custom code before an update.</td>
<td>Valid values are <code>true</code> or <code>false</code>. For more information, see <em>OTA Updates of AWS IoT Greengrass Core Software</em> (p. 148).</td>
</tr>
<tr>
<td><code>writeDirectory</code></td>
<td>Optional. The write directory where AWS IoT Greengrass</td>
<td>For more information, see <em>Configure a Write Directory for AWS IoT Greengrass</em> (p. 52).</td>
</tr>
</tbody>
</table>
### AWS IoT Greengrass Core Configuration File

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>creates all read-write resources.</td>
<td></td>
</tr>
</tbody>
</table>

### Deprecated versions

The following versions of the AWS IoT Greengrass Core software are not supported. This information is included for reference purposes only.

**GGC v1.6**

```json
{
    "coreThing": {
        "caPath": "root-ca-pem",
        "certPath": "cloud-pem-crt",
        "keyPath": "cloud-pem-key",
        "thingArn": "arn:aws:iot:region:account-id:thing/core-thing-name",
        "iotHost": "host-prefix.iot.region.amazonaws.com",
        "ggHost": "greengrass.iot.region.amazonaws.com",
        "keepAlive": 600,
        "mqttMaxConnectionRetryInterval": 60
    },
    "runtime": {
        "cgroup": {
            "useSystemd": "yes/no"
        },
        "managedRespawn": true,
        "writeDirectory": "/write-directory"
    }
}
```

**Note**

If you use the **Easy group creation** option from the AWS IoT Greengrass console, then the `config.json` file is deployed to the core device in a working state that specifies the default configuration.

The `config.json` file supports the following properties:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>caPath</td>
<td>The path to the AWS IoT root CA relative to the <code>/greengrass-root/certs</code> directory.</td>
<td>Save the file under <code>/greengrass-root/certs</code></td>
</tr>
<tr>
<td>certPath</td>
<td>The path to the AWS IoT Greengrass core certificate relative to the <code>/greengrass-root/certs</code> directory.</td>
<td>Save the file under <code>/greengrass-root/certs</code></td>
</tr>
<tr>
<td>keyPath</td>
<td>The path to the AWS IoT Greengrass core private key relative to <code>/greengrass-root/certs</code> directory.</td>
<td>Save the file under <code>/greengrass-root/certs</code></td>
</tr>
<tr>
<td>thingArn</td>
<td>The Amazon Resource Name (ARN) of the AWS IoT thing</td>
<td>Find this for your core in the AWS IoT Greengrass console</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>that represents the AWS IoT Greengrass core device.</td>
<td>under <strong>Cores</strong>, or by running the <code>aws greengrass get-core-definition-version</code> CLI command.</td>
</tr>
<tr>
<td>iotHost</td>
<td>Your AWS IoT endpoint.</td>
<td>Find this in the AWS IoT console under <strong>Settings</strong>, or by running the <code>aws iot describe-endpoint</code> CLI command.</td>
</tr>
<tr>
<td>ggHost</td>
<td>Your AWS IoT Greengrass endpoint.</td>
<td>This value uses the format <code>greengrass.iot.region.amazonaws.com</code>. Use the same region as <code>iotHost</code>.</td>
</tr>
<tr>
<td>keepAlive</td>
<td>The MQTT KeepAlive period, in seconds.</td>
<td>This is an optional value. The default is 600.</td>
</tr>
<tr>
<td>mqttMaxConnectionRetryInterval</td>
<td>The maximum interval (in seconds) between MQTT connection retries if the connection is dropped.</td>
<td>Specify this value as an unsigned integer. This is an optional value. The default is 60.</td>
</tr>
<tr>
<td>useSystemd</td>
<td>Indicates whether your device uses systemd.</td>
<td>Valid values are <code>yes</code> or <code>no</code>. Run the <code>check_ggc_dependencies</code> script in Module 1 (p. 65) to see if your device uses systemd.</td>
</tr>
<tr>
<td>managedRespawn</td>
<td>An optional over-the-air (OTA) updates feature, this indicates that the OTA agent needs to run custom code before an update.</td>
<td>Valid values are <code>true</code> or <code>false</code>. For more information, see OTA Updates of AWS IoT Greengrass Core Software (p. 148).</td>
</tr>
<tr>
<td>writeDirectory</td>
<td>The write directory where AWS IoT Greengrass creates all read-write resources.</td>
<td>This is an optional value. For more information, see Configure a Write Directory for AWS IoT Greengrass (p. 52).</td>
</tr>
</tbody>
</table>

**GGC v1.5**

```json
{
    "coreThing": {
        "caPath": "root-ca-pem",
        "certPath": "cloud-pem-crt",
        "keyPath": "cloud-pem-key",
        "thingArn": "arn:aws:iot:region:account-id:thing/core-thing-name",
        "iotHost": "host-prefix.iot.region.amazonaws.com",
        "ggHost": "greengrass.iot.region.amazonaws.com",
        "keepAlive": 600
    },
    "runtime": {
        "cgroup": {
```
The `config.json` file exists in `/greengrass-root/config` and contains the following parameters:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>caPath</td>
<td>The path to the AWS IoT root CA relative to the <code>/greengrass-root/certs</code> folder.</td>
<td>Save the file under the <code>/greengrass-root/certs</code> folder.</td>
</tr>
<tr>
<td>certPath</td>
<td>The path to the AWS IoT Greengrass core certificate relative to the <code>/greengrass-root/certs</code> folder.</td>
<td>Save the file under the <code>/greengrass-root/certs</code> folder.</td>
</tr>
<tr>
<td>keyPath</td>
<td>The path to the AWS IoT Greengrass core private key relative to <code>/greengrass-root/certs</code> folder.</td>
<td>Save the file under the <code>/greengrass-root/certs</code> folder.</td>
</tr>
<tr>
<td>thingArn</td>
<td>The Amazon Resource Name (ARN) of the AWS IoT thing that represents the AWS IoT Greengrass core device.</td>
<td>Find this for your core in the AWS IoT Greengrass console under <code>Cores</code>, or by running the <code>aws greengrass get-core-definition-version</code> CLI command.</td>
</tr>
<tr>
<td>iotHost</td>
<td>Your AWS IoT endpoint.</td>
<td>Find this in the AWS IoT console under <code>Settings</code>, or by running the <code>aws iot describe-endpoint</code> command.</td>
</tr>
<tr>
<td>ggHost</td>
<td>Your AWS IoT Greengrass endpoint.</td>
<td>This value uses the format <code>greengrass.iot.region.amazonaws.com</code>. Use the same region as iotHost.</td>
</tr>
<tr>
<td>keepAlive</td>
<td>The MQTT KeepAlive period, in seconds.</td>
<td>This is an optional value. The default value is 600 seconds.</td>
</tr>
<tr>
<td>useSystemd</td>
<td>Indicates whether your device uses <code>systemd</code>.</td>
<td>Valid values are <code>yes</code> or <code>no</code>. Run the check_ggc_dependencies script in <code>Module 1</code> (p. 65) to see if your device uses <code>systemd</code>.</td>
</tr>
<tr>
<td>managedRespawn</td>
<td>An optional over-the-air (OTA) updates feature, this indicates that the OTA agent</td>
<td>For more information, see OTA Updates of AWS IoT Greengrass Core Software (p. 148).</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>caPath</td>
<td>The path to the AWS IoT root CA relative to the /greengrass-root/certs folder.</td>
<td>Save the file under the /greengrass-root/certs folder.</td>
</tr>
<tr>
<td>certPath</td>
<td>The path to the AWS IoT Greengrass core certificate relative to the /greengrass-root/certs folder.</td>
<td>Save the file under the /greengrass-root/certs folder.</td>
</tr>
<tr>
<td>keyPath</td>
<td>The path to the AWS IoT Greengrass core private key relative to /greengrass-root/certs folder.</td>
<td>Save the file under the /greengrass-root/certs folder.</td>
</tr>
<tr>
<td>thingArn</td>
<td>The Amazon Resource Name (ARN) of the AWS IoT thing that represents the AWS IoT Greengrass core.</td>
<td>You can find this value in the AWS IoT Greengrass console under the definition for your AWS IoT thing.</td>
</tr>
<tr>
<td>iotHost</td>
<td>Your AWS IoT endpoint.</td>
<td>You can find this value in the AWS IoT console under Settings.</td>
</tr>
<tr>
<td>ggHost</td>
<td>Your AWS IoT Greengrass endpoint.</td>
<td>You can find this value in the AWS IoT console under Settings.</td>
</tr>
</tbody>
</table>

**GGC v1.3**

```json
{
  "coreThing": {
    "caPath": "root-ca.pem",
    "certPath": "cloud-pem.crt",
    "keyPath": "cloud-pem-key",
    "thingArn": "arn:aws:iot:region:account-id:thing/core-thing-name",
    "iotHost": "host-prefix.iot.region.amazonaws.com",
    "ggHost": "greengrass.iot.region.amazonaws.com",
    "keepAlive": 600
  },
  "runtime": {
    "cgroup": {"useSystemd": "yes/no"}
  },
  "managedRespawn": true
}
```

The config.json file exists in `/greengrass-root/config` and contains the following parameters:
Field | Description | Notes
--- | --- | ---
keepAlive | The MQTT KeepAlive period, in seconds. | This is an optional value. The default value is 600 seconds.
useSystemd | A binary flag, if your device uses systemd. | Values are yes or no. Use the dependency script in Module 1 (p. 65) to see if your device uses systemd.
managedRespawn | An optional over-the-air (OTA) updates feature, this indicates that the OTA agent needs to run custom code before an update. | For more information, see OTA Updates of AWS IoT Greengrass Core Software (p. 148).

GGC v1.1

```
{
  "coreThing": {
    "caPath": "root-ca.pem",
    "certPath": "cloud-pem.crt",
    "keyPath": "cloud-pem-key",
    "thingArn": "arn:aws:iot:region:account-id:thing/core-thing-name",
    "iotHost": "host-prefix.iot.region.amazonaws.com",
    "ggHost": "greengrass.iot.region.amazonaws.com",
    "keepAlive": 600
  },
  "runtime": {
    "cgroup": {
      "useSystemd": "yes|no"
    }
  }
}
```

The `config.json` file exists in `/greengrass-root/config` and contains the following parameters:

Field | Description | Notes
--- | --- | ---
caPath | The path to the AWS IoT root CA relative to the `/greengrass-root/certs` folder. | Save the file under the `/greengrass-root/certs` folder.
certPath | The path to the AWS IoT Greengrass core certificate relative to the `/greengrass-root/certs` folder. | Save the file under the `/greengrass-root/certs` folder.
keyPath | The path to the AWS IoT Greengrass core private key relative to the | Save the file under the `/greengrass-root/certs` folder.
### AWS IoT Greengrass Core Configuration File

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>thingArn</td>
<td>The Amazon Resource Name (ARN) of the AWS IoT thing that represents the AWS IoT Greengrass core.</td>
<td>You can find this value in the AWS IoT Greengrass console under the definition for your AWS IoT thing.</td>
</tr>
<tr>
<td>iotHost</td>
<td>Your AWS IoT endpoint.</td>
<td>You can find this value in the AWS IoT console under Settings.</td>
</tr>
<tr>
<td>ggHost</td>
<td>Your AWS IoT Greengrass endpoint.</td>
<td>You can find this value in the AWS IoT console under Settings with greengrass prepended.</td>
</tr>
<tr>
<td>keepAlive</td>
<td>The MQTT KeepAlive period, in seconds.</td>
<td>This is an optional value. The default value is 600 seconds.</td>
</tr>
<tr>
<td>useSystemd</td>
<td>A binary flag, if your device uses systemd.</td>
<td>Values are yes or no. Use the dependency script in Module 1 (p. 65) to see if your device uses systemd.</td>
</tr>
</tbody>
</table>

### GGC v1.0

In AWS IoT Greengrass Core v1.0, `config.json` is deployed to `greengrass-root/configuration`.

```json
{
   "coreThing": {
      "caPath": "root-ca-pem",
      "certPath": "cloud-pem-crt",
      "keyPath": "cloud-pem-key",
      "thingArn": "arn:aws:iot:region:account-id:thing/core-thing-name",
      "iotHost": "host-prefix.iot.region.amazonaws.com",
      "ggHost": "greengrass.iot.region.amazonaws.com",
      "keepAlive": 600
   },
   "runtime": {
      "cgroup": {
         "useSystemd": "yes/no"
      }
   }
}
```

The `config.json` file exists in `greengrass-root/configuration` and contains the following parameters:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>caPath</td>
<td>The path to the AWS IoT root CA relative to the <code>greengrass-root/configuration/certs</code> folder.</td>
<td>Save the file under the <code>greengrass-root/configuration/certs</code> folder.</td>
</tr>
</tbody>
</table>
### Endpoints Must Match the Certificate Type

Your AWS IoT and AWS IoT Greengrass endpoints must correspond to the certificate type of the root CA certificate on your device.

If you’re using an Amazon Trust Services (ATS) root CA certificate (preferred), you must use ATS endpoints. ATS endpoints include the `ats` segment, as shown in the following syntax for the AWS IoT endpoint:

```
prefix-ats.iot.region.amazonaws.com
```

In some AWS Regions, AWS IoT Greengrass also currently supports legacy Verisign endpoints and root CA certificates for backward compatibility. For more information, see AWS IoT Greengrass in the Amazon Web Services General Reference.

If you’re using a legacy Verisign root CA certificate, we recommend that you create an ATS endpoint and use an ATS root CA certificate instead. For more information, see Server Authentication in the AWS IoT Developer Guide. Otherwise, make sure to use the corresponding legacy endpoints. For example, the following syntax is used for legacy AWS IoT endpoints:

```plaintext
prefix-iot.region.amazonaws.com
```
prefix.\texttt{iot.\texttt{region}.amazonaws.com}

If your endpoints and certificate type do not match, authentication attempts between AWS IoT and AWS IoT Greengrass fail.

**Endpoints in \texttt{config\.json}**

On an AWS IoT Greengrass core device, in the \texttt{config.json} file, the AWS IoT and AWS IoT Greengrass endpoints are specified in the \texttt{coreThing} object. The \texttt{iotHost} property represents the AWS IoT endpoint. The \texttt{ggHost} property represents the AWS IoT Greengrass endpoint. In the following example snippet, these properties specify ATS endpoints.

```json
{
  "coreThing": {
    ...,
    "iotHost": "abcde1234uwxyz-ats.iot.us-west-2.amazonaws.com",
    "ggHost": "greengrass-ats.iot.us-west-2.amazonaws.com",
    ...
  },
}
```

**AWS IoT endpoint**

You can get your AWS IoT endpoint by running the \texttt{aws iot describe-endpoint} CLI command with the appropriate \texttt{--endpoint-type} parameter.

- To return an ATS signed endpoint, run:

  ```bash
  aws iot describe-endpoint --endpoint-type iot:Data-ATS
  ```

- To return a legacy Verisign signed endpoint, run:

  ```bash
  aws iot describe-endpoint --endpoint-type iot:Data
  ```

**AWS IoT Greengrass endpoint**

Your AWS IoT Greengrass endpoint is your \texttt{iotHost} endpoint with the host prefix replaced by \texttt{greengrass}. For example, the ATS signed endpoint is \texttt{greengrass-ats.iot.\texttt{region}.amazonaws.com}. This uses the same region as your AWS IoT endpoint.

**Connect on Port 443 or Through a Network Proxy**

This feature is available for AWS IoT Greengrass Core v1.7 and later.

AWS IoT Greengrass communicates with AWS IoT 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 an HTTP proxy before connecting to the internet.

To enable communication in these scenarios, AWS IoT Greengrass allows the following configurations:

- **MQTT with TLS client authentication over port 443.** If your network allows connections to port 443, you can configure the core 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.
AWS IoT Greengrass uses the Application Layer Protocol Network (ALPN) TLS extension to enable this connection. As with the default configuration, MQTT over TLS on port 443 uses certificate-based client authentication.

When configured to use a direct connection to port 443, the core supports over-the-air (OTA) updates (p. 148) for AWS IoT Greengrass software. This support requires AWS IoT Greengrass Core v1.9.3 or later.

- **HTTPS communication over port 443.** AWS IoT Greengrass sends HTTPS traffic over port 8443 by default, but you can configure it to use port 443.
- **Connection through a network proxy.** You can configure a network proxy server to act as an intermediary for connecting to the AWS IoT Greengrass core. Only basic authentication and HTTP and HTTPS proxies are supported.

The proxy configuration is passed to user-defined Lambda functions through the `http_proxy`, `https_proxy`, and `no_proxy` environment variables. User-defined Lambda functions must use these passed-in settings to connect through the proxy. Common libraries used by Lambda functions to make connections (such as boto3 or cURL and python `requests` packages) typically use these environment variables by default. If a Lambda function also specifies these same environment variables, AWS IoT Greengrass doesn't override them.

**Important**

Greengrass cores that are configured to use a network proxy don't support OTA updates (p. 148).

### To configure MQTT over port 443

This feature requires AWS IoT Greengrass Core v1.7 or later.

This procedure allows the core to use port 443 for MQTT messaging.

1. Run the following command to stop the AWS IoT Greengrass daemon:

   ```bash
cd /greengrass-root/ggc/core/
sudo ./greengrassd stop
   ```

2. Open `/greengrass-root/config/config.json` for editing as the su user.

3. In the `coreThing` object, add the `iotMqttPort` property and set the value to 443, as shown in the following example.

   ```json
   {
   "coreThing" : {
   "caPath" : "root.ca.pem",
   "certPath" : "12345abcde.cert.pem",
   "keyPath" : "12345abcde.private.key",
   "iotHost" : "abcd123456wxyz-ats.iot.us-west-2.amazonaws.com",
   "iotMqttPort" : 443,
   "ggHost" : "greengrass-ats.iot.us-west-2.amazonaws.com",
   "keepAlive" : 600
   },
   ...
   }
   ```

4. Start the daemon.

   ```bash
cd /greengrass-root/ggc/core/
sudo ./greengrassd start
   ```
To configure HTTPS over port 443

This feature requires AWS IoT Greengrass Core v1.8 or later.

This procedure configures the core to use port 443 for HTTPS communication.

1. Run the following command to stop the AWS IoT Greengrass daemon:

   ```
   cd /greengrass-root/ggc/core/
   sudo ./greengrassd stop
   ```

2. Open `greengrass-root/config/config.json` for editing as the su user.

3. In the `coreThing` object, add the `iotHttpPort` and `ggHttpPort` properties, as shown in the following example.

   ```
   {
     "coreThing": {
       "caPath": "root.ca.pem",
       "certPath": "12345abcde.cert.pem",
       "keyPath": "12345abcde.private.key",
       "iotHost": "abcd123456wxyz-ats.iot.us-west-2.amazonaws.com",
       "iotHttpPort": 443,
       "ggHost": "greengrass-ats.iot.us-west-2.amazonaws.com",
       "ggHttpPort": 443,
       "keepAlive": 600
     },
     ...
   }
   ```

4. Start the daemon.

   ```
   cd /greengrass-root/ggc/core/
   sudo ./greengrassd start
   ```

To configure a network proxy

This feature requires AWS IoT Greengrass Core v1.7 or later.

This procedure allows AWS IoT Greengrass to connect to the internet through an HTTP or HTTPS network proxy.

1. Run the following command to stop the AWS IoT Greengrass daemon:

   ```
   cd /greengrass-root/ggc/core/
   sudo ./greengrassd stop
   ```

2. Open `greengrass-root/config/config.json` for editing as the su user.

3. In the `coreThing` object, add the `networkProxy` (p. 49) object, as shown in the following example.

   ```
   {
     "coreThing": {
   ```
"caPath" : "root.ca.pem",
"certPath" : "12345abcde.cert.pem",
"keyPath" : "12345abcde.private.key",
"iotHost" : "abcd123456wxyz-ats.iot.us-west-2.amazonaws.com",
"ggHost" : "greengrass-ats.iot.us-west-2.amazonaws.com",
"keepAlive" : 600,
"networkProxy": {
    "proxy" : {
        "url" : "https://my-proxy-server:1100",
        "username" : "Mary_Major",
        "password" : "pass@word1357"
    }
},
...
}

4. Start the daemon.

   cd /greengrass-root/ggc/core/
   sudo ./greengrassd start

**networkProxy object**

Use the `networkProxy` object to specify information about the network proxy. This object has the following properties.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>noProxyAddresses</td>
<td>Optional. A comma-separated list of IP addresses or host names that are exempt from the proxy.</td>
</tr>
<tr>
<td>proxy</td>
<td>The proxy to connect to. A proxy has the following properties.</td>
</tr>
<tr>
<td></td>
<td>• url. The URL of the proxy server, in the format scheme://userinfo@host:port.</td>
</tr>
<tr>
<td></td>
<td>• scheme. The scheme. Must be http or https.</td>
</tr>
<tr>
<td></td>
<td>• userinfo. Optional. The user name and password information. If specified, the username and password fields are ignored.</td>
</tr>
<tr>
<td></td>
<td>• host. The host name or IP address of the proxy server.</td>
</tr>
<tr>
<td></td>
<td>• port. Optional. The port number. If not specified, the following default values are used:</td>
</tr>
<tr>
<td></td>
<td>• http: 80</td>
</tr>
<tr>
<td></td>
<td>• https:443</td>
</tr>
<tr>
<td></td>
<td>• username. Optional. The user name to use to authenticate to the proxy server.</td>
</tr>
<tr>
<td></td>
<td>• password. Optional. The password to use to authenticate to the proxy server.</td>
</tr>
</tbody>
</table>
Whitelisting Endpoints

Communication between Greengrass devices and AWS IoT or AWS IoT Greengrass must be authenticated. This authentication is based on registered X.509 device certificates and cryptographic keys. To allow authenticated requests to pass through proxies without additional encryption, whitelist the following endpoints.

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Port</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>greengrass.region.amazonaws.com</td>
<td>443</td>
<td>Used for control plane operations for group management.</td>
</tr>
<tr>
<td>prefix-ats.iot.region.amazonaws.com or prefix.iot.region.amazonaws.com</td>
<td>MQTT: 8883 or 443 or HTTPS: 8443 or 443</td>
<td>Used for data plane operations for device management, such as shadow sync.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Whitelist one or both endpoints, depending on whether your core and connected devices use Amazon Trust Services (preferred) root CA certificates, legacy root CA certificates, or both. For more information, see the section called “Endpoints Must Match the Certificate Type” (p. 45).</td>
</tr>
<tr>
<td>greengrass-ats.iot.region.amazonaws.com or greengrass.iot.region.amazonaws.com</td>
<td>8443 or 443</td>
<td>Used for device discovery operations.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Whitelist one or both endpoints, depending on whether your core and connected devices use Amazon Trust Services (preferred) root CA certificates, legacy root CA certificates, or both. For more information, see the section called “Endpoints Must Match the Certificate Type” (p. 45).</td>
</tr>
</tbody>
</table>
Connect on Port 443 or Through a Network Proxy

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Port</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>connected devices use Amazon Trust Services (preferred) root CA certificates, legacy root CA certificates, or both. For more information, see the section called “Endpoints Must Match the Certificate Type” (p. 45).</td>
</tr>
</tbody>
</table>

**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.
### Configure a Write Directory for AWS IoT Greengrass

This feature is available for AWS IoT Greengrass Core v1.6 and later.

By default, the AWS IoT Greengrass Core software is deployed under a single root directory where AWS IoT Greengrass performs all read and write operations. However, you can configure AWS IoT Greengrass to use a separate directory for all write operations, including creating directories and files. In this case, AWS IoT Greengrass uses two top-level directories:

- The `greengrass-root` directory, which you can leave as read-write or optionally make read-only. This contains the AWS IoT Greengrass Core software and other critical components that should remain immutable during runtime, such as certificates and `config.json`.
- The specified write directory. This contains writable content, such as logs, state information, and deployed user-defined Lambda functions.

This configuration results in the following directory structure.

**Greengrass root directory**

```
  greengrass-root/
  |-- certs/
  |  |-- root.ca.pem
  |  |-- hash.cert.pem
  |  |-- hash.private.key
  |  |-- hash.public.key
  |-- config/
  |  |-- config.json
  |-- ggc/
  |-- packages/
  |  |-- package-version/
  |     |-- bin/
```
Write Directory

```
|write-directory/
|-- packages/
 | |-- package-version/
 |    |-- ggc_root/
 |    |-- rootfs_nosys/
 |    |-- rootfs_sys/
 |    |-- var/
|-- deployment/
 | |-- group/
 |    |-- group.json
 | |-- lambda/
 | |-- mlmodel/
|-- var/
 | |-- log/
 | |-- state/
```

To configure a write directory

1. Run the following command to stop the AWS IoT Greengrass daemon:

   ```
cd /greengrass-root/ggc/core/
sudo ./greengrassd stop
   ```

2. Open `/greengrass-root/config/config.json` for editing as the su user.

3. Add `writeDirectory` as a parameter and specify the path to the target directory, as shown in the following example.

   ```
   {
     "coreThing": {
       "caPath": "root-CA.pem",
       "certPath": "hash.pem.crt",
       ...
     },
     ...
     "writeDirectory" : "/write-directory"
   }
   ```

**Note**

You can update the `writeDirectory` setting as often as you want. After the setting is updated, AWS IoT Greengrass uses the newly specified write directory at the next start, but doesn't migrate content from the previous write directory.

4. Now that your write directory is configured, you can optionally make the `greengrass-root` directory read-only. For instructions, see To Make the Greengrass Root Directory Read-Only (p. 54).
Otherwise, start the AWS IoT Greengrass daemon:

```
cd /greengrass-root/ggc/core/
sudo ./greengrassd start
```

To make the Greengrass root directory read-only

Follow these steps only if you want to make the Greengrass root directory read-only. The write directory must be configured before you begin.

1. Grant access permissions to required directories:
   a. Give read and write permissions to the config.json owner.
      
      ```
sudo chmod 0600 /greengrass-root/config/config.json
      ```
   b. Make ggc_user the owner of the certs and system Lambda directories.
      
      ```
sudo chown -R ggc_user:ggc_group /greengrass-root/certs/
sudo chown -R ggc_user:ggc_group /greengrass-root/ggc/packages/1.9.4/lambda/
      ```

   **Note**
   The ggc_user and ggc_group accounts are used by default to run system Lambda functions. If you configured the group-level default access identity (p. 183) to use different accounts, you should give permissions to that user (UID) and group (GID) instead.

2. Make the greengrass-root directory read-only by using your preferred mechanism.

   **Note**
   One way to make the greengrass-root directory read-only is to mount the directory as read-only. However, to apply over-the-air (OTA) updates to the AWS IoT Greengrass Core software in a mounted directory, the directory must first be unmounted, and then remounted after the update. You can add these umount and mount operations to the ota_pre_update and ota_post_update scripts. For more information about OTA updates, see the section called “Greengrass OTA Update Agent” (p. 149) and the section called “AWS IoT Greengrass Core Update with Managed Respawn” (p. 152).

3. Start the daemon.

   ```
   cd /greengrass-root/ggc/core/
sudo ./greengrassd start
   ```

   If the permissions from step 1 aren't set correctly, then the daemon won't start.

Message Quality of Service

In the AWS IoT Greengrass environment, local devices, Lambda functions, connectors, and system components can communicate with each other and with the cloud. All communication goes through the core. Messages destined for local targets use a quality of service (QoS) level that is different from messages destined for the cloud.
• **Messages with local targets use QoS 0.** The core makes one attempt to send a message to its target. It doesn't store messages or confirm delivery. Messages can be dropped anywhere between components.

  **Note**
  Although direct communication between Lambda functions doesn't use MQTT messaging, the behavior is the same.

• **Messages with cloud targets use QoS 1.** The core sends these messages to the spooler system component, which sends them to the cloud using QoS 1. This allows the spooler to manage the MQTT message queue (p. 55). If message delivery isn't confirmed by AWS IoT, the message is spooled to be retried later (unless the queue is full). For more information, see the section called “MQTT Message Queue” (p. 55).

  **Note**
  Although QoS 1 is used internally to manage the queue, message publishers can send MQTT messages with QoS 0 only.

**MQTT Message Queue**

MQTT messages that are destined for cloud targets are queued to await processing. Queued messages are processed in first in first out (FIFO) order. After a message is processed and published to the cloud, the message is removed from the queue. AWS IoT Greengrass manages the queue by using a system GGCloudSpooler Lambda function.

**Configure the MQTT Message Queue**

This feature is available for AWS IoT Greengrass Core v1.6 and later.

You can configure AWS IoT Greengrass to store unprocessed messages in memory or in a local storage cache. Unlike in-memory storage, the local storage cache has the ability to persist across core restarts (for example, after a group deployment or a device reboot), so AWS IoT Greengrass can continue to process the messages. You can also configure the storage size.

  **Note**
  Versions 1.5.0 and earlier use in-memory storage with a queue size of 2.5 MB. You cannot configure storage settings for earlier versions.

The following environment variables for the GGCloudSpooler Lambda function are used to define storage settings.

• **GG_CONFIG_STORAGE_TYPE.** The location of the message queue. The following are valid values:
  - FileSystem. Store unprocessed messages in the local storage cache. When the core restarts, queued messages are retained for processing. Messages are removed after they are processed.
  - Memory (default). Store unprocessed messages in memory. When the core restarts, queued messages are lost.

    This option is optimized for devices with restricted hardware capabilities. When using this configuration, we recommend that you deploy groups or restart the device when the service disruption is the lowest.

• **GG_CONFIG_MAX_SIZE_BYTES.** The storage size, in bytes. This value can be any non-negative integer greater than or equal to 262144 (256 KB); a smaller size prevents the AWS IoT Greengrass Core software from starting. The default size is 2.5 MB. When the size limit is reached, the oldest queued messages are replaced by new messages.
To Cache Messages in Local Storage

To configure AWS IoT Greengrass to cache messages to the file system so they persist across core restarts, you create a function definition version where the GGCloudSpooler function specifies the FileSystem storage type. You must use the AWS IoT Greengrass API to configure the local storage cache. You can’t do this in the console.

The following procedure uses the `create-function-definition-version` CLI command to configure the spooler to save queued messages to the file system. It also configures a 2.6 MB queue size.

1. Get the IDs of the target Greengrass group and group version. In this procedure, we assume that you’re updating the configuration of the latest group and group version. The following command returns the most recently created group.

   ```bash
   aws greengrass list-groups --query "reverse(sort_by(Groups, &CreationTimestamp))[0]"
   ``

   Or, you can query by name. Group names are not required to be unique, so multiple groups might be returned.

   ```bash
   aws greengrass list-groups --query "Groups[?Name=='MyGroup']"
   ``

   **Note**
   You can also find these values in the AWS IoT console. The group ID is shown on the group’s Settings page. Group version IDs are shown on the group’s Deployments page.

2. Copy the Id and LatestVersion values from the target group in the output.

3. Get the latest group version.
   - Replace `group-id` with the Id that you copied.
   - Replace `latest-group-version-id` with the LatestVersion that you copied.

   ```bash
   aws greengrass get-group-version
   --group-id group-id
   --group-version-id latest-group-version-id
   ``

4. From the Definition object in the output, copy the CoreDefinitionVersionArn and the ARNs of all other group components except FunctionDefinitionVersionArn. You use these values when you create a new group version.

5. From the FunctionDefinitionVersionArn in the output, copy the ID of the function definition. The ID is the GUID that follows the functions segment in the ARN, as shown in the following example.

   ```bash
   arn:aws:greengrass:us-west-2:123456789012:/greengrass/definition/functions/bcfc6b49-beb0-4396-b703-6dEXAMPLEcu5/versions/0f7337b4-922b-45c5-856f-1aEXAMPLEsf6
   ``

   **Note**
   You can optionally create a function definition by running the `create-function-definition` command, and then copy the ID from the output.

6. Add a function definition version to the function definition.
   - Replace `function-definition-id` with the Id that you copied for the function definition.
   - Replace `arbitrary-function-id` with a name for the function, such as `spooler-function`.
   - Add any Lambda functions that you want to include in this version to the functions array. You can use the `get-function-definition-version` command to get the Greengrass Lambda functions from an existing function definition version.
Warning
Make sure that you specify a value for `GG_CONFIG_MAX_SIZEBYTES` that's greater than or equal to 262144. A smaller size prevents the AWS IoT Greengrass Core software from starting.

```
aws greengrass create-function-definition-version
  --function-definition-id function-definition-id
  --functions '[{"FunctionArn":
  "arn:aws:lambda:::function:GGCloudSpooler:1","FunctionConfiguration": {
  "Environment": {
    "Variables": {
      "GG_CONFIG_MAX_SIZEBYTES": "2621440", "GG_CONFIG_STORAGE_TYPE": "FileSystem"},
    "Executable": "spooler","MemorySize": 32768,"Pinned": true,"Timeout": 3},"Id": "arbitrary-function-id"}]
```

7. Copy the Arn of the function definition version from the output.

8. Create a group version that contains the system Lambda function.
   - Replace `group-id` with the Id for the group.
   - Replace `core-definition-version-arn` with the CoreDefinitionVersionArn that you copied from the latest group version.
   - Replace `function-definition-version-arn` with the Arn that you copied for the new function definition version.
   - Replace the ARNs for other group components (for example, SubscriptionDefinitionVersionArn or DeviceDefinitionVersionArn) that you copied from the latest group version.
   - Remove any unused parameters. For example, remove the `--resource-definition-version-arn` if your group version doesn't contain any resources.

```
aws greengrass create-group-version
  --group-id group-id
  --core-definition-version-arn core-definition-version-arn
  --function-definition-version-arn function-definition-version-arn
  --device-definition-version-arn device-definition-version-arn
  --logger-definition-version-arn logger-definition-version-arn
  --resource-definition-version-arn resource-definition-version-arn
  --subscription-definition-version-arn subscription-definition-version-arn
```

9. Copy the Version from the output. This is the ID of the new group version.

10. Deploy the group with the new group version.
    - Replace `group-id` with the Id that you copied for the group.
    - Replace `group-version-id` with the Version that you copied for the new group version.

```
aws greengrass create-deployment
  --group-id group-id
  --group-version-id group-version-id
  --deployment-type NewDeployment
```

To update the storage settings, you use the AWS IoT Greengrass API to create a new function definition version that contains the GGCloudSpooler function with the updated configuration. Then add the function definition version to a new group version (along with your other group components) and deploy the group version. If you want to restore the default configuration, you can create a function definition version that doesn't include the GGCloudSpooler function.
This system Lambda function isn't visible in the console. However, after the function is added to the latest group version, it's included in deployments that you make from the console (unless you use the API to replace or remove it).

Client IDs for MQTT Connections with AWS IoT

This feature is available for AWS IoT Greengrass Core v1.8 and later.

The AWS IoT Greengrass core opens MQTT connections with AWS IoT for operations such as shadow sync and certificate management. For these connections, the core generates predictable client IDs based on the core thing name. Predictable client IDs can be used with monitoring, auditing, and pricing features, including AWS IoT Device Defender and AWS IoT lifecycle events. You can also create logic around predictable client IDs (for example, subscribe policy templates based on certificate attributes).

GGC v1.9

Two Greengrass system components open MQTT connections with AWS IoT. These components use the following patterns to generate the client IDs for the connections.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Client ID pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deployments</td>
<td><code>core-thing-name</code></td>
</tr>
<tr>
<td>Example: MyCoreThing</td>
<td>Use this client ID for connect, disconnect, subscribe, and unsubscribe lifecycle event notifications.</td>
</tr>
<tr>
<td>MQTT message exchange with AWS IoT</td>
<td><code>core-thing-name-cnn</code></td>
</tr>
<tr>
<td>Example: MyCoreThing-c01</td>
<td><code>nn</code> is an integer that starts at 00 and increments with each new connection to a maximum of 05. The number of connections is determined by the number of devices that sync their shadow state with AWS IoT (maximum 200 per group) and the number of subscriptions with <code>cloud</code> as their source in the group (maximum 50 per group).</td>
</tr>
<tr>
<td></td>
<td>The spooler system component makes connections with AWS IoT to exchange messages for subscriptions with a cloud source or target. The spooler also acts as proxy for message exchange between AWS IoT and the local shadow service and device certificate manager.</td>
</tr>
</tbody>
</table>

GGC v1.8

Several Greengrass system components open MQTT connections with AWS IoT. These components use the following patterns to generate the client IDs for the connections.
Activate Automatic IP Detection

You can configure AWS IoT Greengrass to enable automatic discovery of your AWS IoT Greengrass core using the IPDetector system Lambda function. This feature can also be enabled by choosing Automatic detection when deploying your group from the console for the first time, or from the group settings page in the console at any time.

The following procedure uses the create-function-definition-version CLI command to configure automatic discovery of the Greengrass core.

1. Get the IDs of the target Greengrass group and group version. In this procedure, we assume that you’re updating the configuration of the latest group and group version. The following command returns the most recently created group.

   ```bash
   aws greengrass list-groups --query "reverse(sort_by(Groups, &CreationTimestamp))[0]"
   ```
Or, you can query by name. Group names are not required to be unique, so multiple groups might be returned.

```
aws greengrass list-groups --query "Groups[?Name=='MyGroup']"
```

**Note**
You can also find these values in the AWS IoT console. The group ID is shown on the group's Settings page. Group version IDs are shown on the group's Deployments page.

2. Copy the Id and LatestVersion values from the target group in the output.

3. Get the latest group version.
   - Replace `group-id` with the Id that you copied.
   - Replace `latest-group-version-id` with the LatestVersion that you copied.

```
aws greengrass get-group-version \
--group-id group-id \ 
--group-version-id latest-group-version-id
```

4. From the Definition object in the output, copy the CoreDefinitionVersionArn and the ARNs of all other group components except FunctionDefinitionVersionArn. You use these values when you create a new group version.

5. From the FunctionDefinitionVersionArn in the output, copy the ID of the function definition and the function definition version:

```
arn:aws:greengrass:region:account-id:/greengrass/groups/function-definition-id/versions/function-definition-version-id
```

**Note**
You can optionally create a function definition by running the `create-function-definition` command, and then copy the ID from the output.

6. Use the `get-function-definition-version` command to get the current definition state. Use the `function-definition-id` you copied for the function definition. For example, `4d941bc7-92a1-4f45-8d64-EXAMPLEf76c3`.

```
aws greengrass get-function-definition-version \ 
--function-definition-id function-definition-id \ 
--function-definition-version-id function-definition-version-id
```

Make a note of the listed function configurations. You will need to include these when creating a new function definition version in order to prevent loss of your current definition settings.

7. Add a function definition version to the function definition.
   - Replace `function-definition-id` with the Id that you copied for the function definition. For example, `4d941bc7-92a1-4f45-8d64-EXAMPLEf76c3`.
   - Replace `arbitrary-function-id` with a name for the function, such as `auto-detection-function`.
   - Add all Lambda functions that you want to include in this version to the `functions` array, such as any listed in the previous step.

```
aws greengrass create-function-definition-version \
```


Activate Automatic IP Detection

```
--function-definition-id function-definition-id \
--functions '[["FunctionArn":"arn:aws:lambda:::function:GGIPDetector:1","Id":"arbitrary-function-id","FunctionConfiguration":{"Pinned":true,"MemorySize":32768,"Timeout":3}]\'
--region us-west-2
```

8. Copy the Arn of the function definition version from the output.

9. Create a group version that contains the system Lambda function.
    - Replace `group-id` with the Id for the group.
    - Replace `core-definition-version-arn` with the CoreDefinitionVersionArn that you copied from the latest group version.
    - Replace `function-definition-version-arn` with the Arn that you copied for the new function definition version.
    - Replace the ARNs for other group components (for example, SubscriptionDefinitionVersionArn or DeviceDefinitionVersionArn) that you copied from the latest group version.
    - Remove any unused parameters. For example, remove the `--resource-definition-version-arn` if your group version doesn't contain any resources.

```
aws greengrass create-group-version \
  --group-id group-id \
  --core-definition-version-arn core-definition-version-arn \
  --function-definition-version-arn function-definition-version-arn \
  --device-definition-version-arn device-definition-version-arn \
  --logger-definition-version-arn logger-definition-version-arn \
  --resource-definition-version-arn resource-definition-version-arn \
  --subscription-definition-version-arn subscription-definition-version-arn
```

10. Copy the Version from the output. This is the ID of the new group version.

11. Deploy the group with the new group version.
    - Replace `group-id` with the Id that you copied for the group.
    - Replace `group-version-id` with the Version that you copied for the new group version.

```
aws greengrass create-deployment \
  --group-id group-id \
  --group-version-id group-version-id \
  --deployment-type NewDeployment
```

If you want to manually input the IP address of your AWS IoT Greengrass core, you can complete this tutorial with a different function definition that does not include the IPDetector function. This will prevent the detection function from locating and automatically inputting your AWS IoT Greengrass core IP address.

This system Lambda function isn't visible in the Lambda console. After the function is added to the latest group version, it's included in deployments that you make from the console (unless you use the API to replace or remove it).
Configure the Init System to Start the Greengrass Daemon

It's a good practice to set up your init system to start the Greengrass daemon during boot, especially when managing large fleets of devices.

There are different types of init system, such as initd, systemd, and SystemV, and they use similar configuration parameters. The following example is a service file for systemd. The Type parameter is set to forking because greengrassd (which is used to start Greengrass) forks the Greengrass daemon process, and the Restart parameter is set to on-failure to direct systemd to restart Greengrass if Greengrass enters a failed state.

**Note**
To see if your device uses systemd, run the check_ggc_dependencies script as described in Module 1 (p. 65). Then to use systemd, make sure that the useSystemd parameter in config.json (p. 24) is set to yes.

```
[Unit]
Description=Greengrass Daemon

[Service]
Type=forking
PIDFile=/var/run/greengrassd.pid
Restart=on-failure
ExecStart=/greengrass/ggc/core/greengrassd start
ExecReload=/greengrass/ggc/core/greengrassd restart
ExecStop=/greengrass/ggc/core/greengrassd stop

[Install]
WantedBy=multi-user.target
```

For information about how to create and enable a service file for systemd on a Raspberry Pi, see SYSTEMD in the Raspberry Pi documentation.

Archive an AWS IoT Greengrass Core Software Installation

When you upgrade to a new version of the AWS IoT Greengrass Core software, you can archive the currently installed version. This preserves your current installation environment so you can test a new software version on the same hardware. This also makes it easy to roll back to your archived version for any reason.

**To archive the current installation and install a new version**

1. Download the AWS IoT Greengrass Core Software (p. 16) installation package that you want to upgrade to.
2. Copy the package to the destination core device. For instructions that show how to transfer files, see this step (p. 81).

   **Note**
   You copy your current certificates, keys, and configuration file to the new installation later.

   Run the commands in the following steps in your core device terminal.
3. Make sure that the Greengrass daemon is stopped on the core device.
   a. To check whether the daemon is running:
If the output contains a root entry for `/greengrass/ggc/packages/ggc-version/bin/daemon`, then the daemon is running.

**Note**
This procedure is written with the assumption that the AWS IoT Greengrass Core software is installed in the `/greengrass` directory.

b. To stop the daemon:

```bash
cd /greengrass/ggc/core/
sudo ./greengrassd stop
```

4. Move the current Greengrass root directory to a different directory.

```bash
sudo mv /greengrass /greengrass_backup
```

5. Untar the new software on the core device. Replace the `os-architecture` and `version` placeholders in the command.

```bash
sudo tar -zxvf greengrass-os-architecture-version.tar.gz -C /
```

6. Copy the archived certificates, keys, and configuration file to the new installation.

```bash
sudo cp /greengrass_backup/certs/* /greengrass/certs
sudo cp /greengrass_backup/config/* /greengrass/config
```

7. Start the daemon:

```bash
cd /greengrass/ggc/core/
sudo ./greengrassd start
```

Now, you can make a group deployment to test the new installation. If something fails, you can restore the archived installation.

**To restore the archived installation**

1. Stop the daemon.
2. Delete the new `/greengrass` directory.
3. Move the `/greengrass_backup` directory back to `/greengrass`.
4. Start the daemon.

**See Also**

- the section called “Hardware Security” (p. 448)
Getting Started with AWS IoT Greengrass

This tutorial includes several modules designed to show you AWS IoT Greengrass basics and help you get started quickly. This tutorial covers:

- The AWS IoT Greengrass programming model.
- Fundamental concepts, such as AWS IoT Greengrass cores, groups, and subscriptions.
- The deployment process for running AWS Lambda functions at the edge.

Requirements

To complete this tutorial, you need the following:

- A Mac, Windows PC, or UNIX-like system.
- An Amazon Web Services (AWS) account. If you don’t have one, see the section called “Create an AWS Account” (p. 65).
- The use of an AWS Region that supports AWS IoT Greengrass. For the list of supported regions for AWS IoT Greengrass, see AWS Regions and Endpoints in the AWS General Reference.

  **Important**
  Make a note of your AWS Region to ensure that it is consistently used throughout this tutorial. Inadvertently switching AWS Regions during the tutorial can create problems. The last exercise in this tutorial is written with the assumption that you are using the US East (N. Virginia) Region.

- A Raspberry Pi 4 Model B, or Raspberry Pi 3 Model B/B+, with a 8 GB microSD card, or an Amazon EC2 instance. Because AWS IoT Greengrass should ideally be used with physical hardware, we recommend that you use a Raspberry Pi.

  **Note**
  Run the following command to get the model of your Raspberry Pi:

  ```bash
  cat /proc/cpuinfo
  ```

  Near the bottom of the listing, make a note of the value of the Revision attribute and then consult the Which Pi have I got? table. For example, if the value of Revision is a02082, the table shows the Pi is a 3 Model B.
  Run the following command to determine the architecture of your Raspberry Pi:

  ```bash
  uname -m
  ```

  For this tutorial, the result should be greater than or equal to armv71.

- Basic familiarity with Python.

Although this tutorial is intended to run AWS IoT Greengrass on a Raspberry Pi, AWS IoT Greengrass also supports other platforms. For more information, see the section called “Supported Platforms and Requirements” (p. 11).
Create an AWS Account

If you don’t have an AWS account, follow these steps:

1. Open the AWS home page, and choose Create an AWS Account.
   
   **Note**
   
   If you've signed in to AWS recently, you might see Sign In to the Console instead.

2. Follow the online instructions. Part of the sign-up procedure involves receiving a phone call and entering a PIN using your phone keypad.
   
   **Important**
   
   Before you begin, make sure that your account has administrative permissions.

Module 1: Environment Setup for Greengrass

This module shows you how to get an out-of-the-box Raspberry Pi, Amazon EC2 instance, or other device ready to be used by AWS IoT Greengrass as your AWS IoT Greengrass core device.

Before you begin, read the requirements (p. 64) for this tutorial.

This module should take less than 30 minutes to complete.

**Topics**

- Setting Up a Raspberry Pi (p. 65)
- Setting Up an Amazon EC2 Instance (p. 70)
- Setting Up Other Devices (p. 75)

**Note**

To learn how to use AWS IoT Greengrass running in a prebuilt Docker container, see the section called “Run AWS IoT Greengrass in a Docker Container” (p. 189).

Setting Up a Raspberry Pi

If you are setting up a Raspberry Pi for the first time, you must follow all of these steps. Otherwise, you can skip to step 9. However, we recommend that you re-image your Raspberry Pi with the operating system as recommended in step 2.

1. Download and install an SD card formatter such as SD Memory Card Formatter or PiBakery. Insert the SD card into your computer. Start the program and choose the drive where you have inserted your SD card. You can perform a quick format of the SD card.

2. Download the Raspbian Buster operating system as a zip file.

3. Using an SD card-writing tool (such as Etcher), follow the tool’s instructions to flash the downloaded zip file onto the SD card. Because the operating system image is large, this step might take some time. Eject your SD card from your computer, and insert the microSD card into your Raspberry Pi.

4. For the first boot, we recommend that you connect the Raspberry Pi to a monitor (through HDMI), a keyboard, and a mouse. Next, connect your Pi to a micro USB power source and the Raspbian operating system should start up.

5. You might want to configure the Pi’s keyboard layout before you continue. To do so, choose the Raspberry icon in the upper-right, choose Preferences and then choose Mouse and Keyboard Settings. Next, on the Keyboard tab, choose Keyboard Layout, and then choose an appropriate keyboard variant.
6. Next, connect your Raspberry Pi to the internet through a Wi-Fi network or an Ethernet cable.

   **Note**
   Connect your Raspberry Pi to the same network that your computer is connected to, and be sure that both your computer and Raspberry Pi have internet access before you continue. If you're in a work environment or behind a firewall, you might need to connect your Pi and your computer to the guest network to get both devices on the same network. However, this approach might disconnect your computer from local network resources, such as your intranet. One solution is to connect the Pi to the guest Wi-Fi network and to connect your computer to the guest Wi-Fi network and your local network through an Ethernet cable. In this configuration, your computer should be able to connect to the Raspberry Pi through the guest Wi-Fi network and your local network resources through the Ethernet cable.

7. You must set up SSH on your Pi to remotely connect to it. On your Raspberry Pi, open a terminal window and run the following command:

```
sudo raspi-config
```

You should see the following:

![Raspberry Pi Software Configuration Tool (raspi-config)](image)

Scroll down and choose **Interfacing Options** and then choose P2 SSH. When prompted, choose Yes. (Use the Tab key followed by Enter). SSH should now be enabled. Choose OK. Use the Tab key to choose Finish and then press Enter. If the Raspberry Pi doesn't reboot automatically, run the following command:

```
sudo reboot
```

8. On your Raspberry Pi, run the following command in the terminal:

```
hostname -I
```

This returns the IP address of your Raspberry Pi.

   **Note**
   For the following, if you receive an ECDSA key fingerprint message (Are you sure you want to continue connecting (yes/no)?), enter yes. The default password for the Raspberry Pi is raspberry.

If you are using macOS, open a terminal window and enter the following:

```
ssh pi@IP-address
```
**IP-address** is the IP address of your Raspberry Pi that you obtained by using the `hostname -I` command.

If you are using Windows, you need to install and configure PuTTY. Expand **Connection**, choose **Data**, and make sure that **Prompt** is selected:

Next, choose **Session**, enter the IP address of the Raspberry Pi, and then choose **Open** using default settings.
If a PuTTY security alert is displayed, choose Yes.

The default Raspberry Pi login and password are **pi** and **raspberry**, respectively.
Note
If your computer is connected to a remote network using VPN, you might have difficulty connecting from the computer to the Raspberry Pi using SSH.

9. You are now ready to set up the Raspberry Pi for AWS IoT Greengrass. First, run the following commands from a local Raspberry Pi terminal window or an SSH terminal window:

```bash
sudo adduser --system ggc_user
sudo addgroup --system ggc_group
```

10. To improve security on the Pi device, enable hardlink and softlink (symlink) protection on the operating system at startup.

   ```bash
cd /etc/sysctl.d
ls
```

   **Note**
   If you don’t see the `98-rpi.conf` file, follow the instructions in the `README.sysctl` file.

   b. Use a text editor (such as Leafpad, GNU nano, or vi) to add the following two lines to the end of the file. You might need to use the `sudo` command to edit as root (for example, `sudo nano 98-rpi.conf`).

   ```bash
   fs.protected_hardlinks = 1
   fs.protected_symlinks = 1
   ```

   c. Reboot the Pi.

   ```bash
   sudo reboot
   ```

   After about a minute, connect to the Pi using SSH and then run the following command to confirm the change:

   ```bash
   sudo sysctl -a 2> /dev/null | grep fs.protected
   ```

   You should see `fs.protected_hardlinks = 1` and `fs.protected_symlinks = 1`.

11. Edit your command line boot file to enable and mount memory cgroups. This allows AWS IoT Greengrass to set the memory limit for Lambda functions. Cgroups are also required to run AWS IoT Greengrass in the default containerization (p. 181) mode.
   a. Navigate to your `boot` directory.

   ```bash
cd /boot/
```

   b. Use a text editor to open `cmdline.txt`. Append the following to the end of the existing line, not as a new line.

   ```bash
cgroup_enable=memory cgroup_memory=1
   ```

   c. Now reboot the Pi.

   ```bash
   sudo reboot
   ```
Your Raspberry Pi should now be ready for AWS IoT Greengrass.

12. To make sure that you have all required dependencies, download and run the Greengrass dependency checker from the AWS IoT Greengrass Samples repository on GitHub. These commands unzip and run the dependency checker script in the Downloads directory.

```bash
cd /home/pi/Downloads
mkdir greengrass-dependency-checker-GGCv1.9.x
cd greengrass-dependency-checker-GGCv1.9.x
unzip greengrass-dependency-checker-GGCv1.9.x.zip
cd greengrass-dependency-checker-GGCv1.9.x
sudo modprobe configs
sudo ./check_ggc_dependencies | more
```

Where more appears, press the Spacebar key to display another screen of text.

**Important**
This tutorial requires Python 2.7. The check_ggc_dependencies script might produce warnings about the missing optional Node.js and Java prerequisites. You can ignore these warnings.

For information about the modprobe command, run `man modprobe` in the terminal.

Your Raspberry Pi configuration is complete. Continue to the section called “Module 2: Installing the AWS IoT Greengrass Core Software” (p. 76).

### Setting Up an Amazon EC2 Instance

This section provides instructions for setting up your Amazon EC2 instance.

**Note**
Although you can complete these modules using an Amazon EC2 instance, AWS IoT Greengrass should ideally be used with physical hardware. We recommend that you use a Raspberry Pi for these tutorials.

1. Sign in to the AWS Management Console and launch an Amazon EC2 instance using an Amazon Linux AMI. For information about Amazon EC2 instances, see the Amazon EC2 Getting Started Guide.

2. After your Amazon EC2 instance is running, enable port 8883 to allow incoming MQTT communications so that other devices can connect with the AWS IoT Greengrass core.
   a. In the navigation pane of the Amazon EC2 console, choose Security Groups.
b. Select the security group for the instance that you just launched, and then choose the **Inbound** tab.

c. Choose **Edit**.
To enable port 8883, you add a custom TCP rule to the security group. For more information, see Adding Rules to a Security Group in the Amazon EC2 User Guide for Linux Instances.

d. On the Edit inbound rules page, choose Add Rule, enter the following settings, and then choose Save.

- For Type, choose Custom TCP Rule.
- For Port Range, enter 8883.
- For Source, choose Anywhere.
- For Description, enter MQTT Communications.

3. Connect to your Amazon EC2 instance.

a. In the navigation pane, choose Instances, choose your instance, and then choose Connect.

b. Follow the instructions on the Connect To Your Instance page to connect to your instance by using SSH and your private key file.
You can use PuTTY for Windows or Terminal for macOS. For more information, see Connect to Your Linux Instance in the Amazon EC2 User Guide for Linux Instances.

4. After you are connected to your Amazon EC2 instance, create user ggc_user and group ggc_group:

```
sudo adduser --system ggc_user
sudo groupadd --system ggc_group
```

**Note**
If the adduser command isn't available on your system, use the following command.

```
sudo useradd --system ggc_user
```

5. To improve security on the device, ensure that hardlink and softlink (symlink) protections are enabled on the operating system at start up.

**Note**
The steps for enabling hardlink and softlink protection vary by operating system. Consult the documentation for your distribution.

a. Run the following command to check if hardlink and softlink protections are enabled:

```
sudo sysctl -a | grep fs.protected
```

If hardlinks and softlinks are set to 1, your protections are enabled correctly. Proceed to step 6.
Note
Softlinks are represented by `fs.protected_symlinks`.

b. If hardlinks and softlinks are not set to 1, enable these protections. Navigate to your system configuration file.

```bash
cd /etc/sysctl.d
ls
```

c. Using your favorite text editor (Leafpad, GNU nano, or vi), add the following two lines to the end of the system configuration file. On Amazon Linux 1, this is the `00-defaults.conf` file. On Amazon Linux 2, this is the `99-amazon.conf` file. You might need to change permissions (using the `chmod` command) to write to the file, or use the `sudo` command to edit as root (for example, `sudo nano 00-defaults.conf`).

```bash
fs.protected_hardlinks = 1
fs.protected_symlinks = 1
```

d. Reboot the Amazon EC2 instance.

```bash
sudo reboot
```

After a few minutes, connect to your instance using SSH and then run the following command to confirm the change.

```bash
sudo sysctl -a | grep fs.protected
```

You should see that hardlinks and softlinks are set to 1.

6. Extract and run the following script to mount Linux control groups (`cgroups`). This allows AWS IoT Greengrass to set the memory limit for Lambda functions. Cgroups are also required to run AWS IoT Greengrass in the default containerization (p. 181) mode.

```bash
curl https://raw.githubusercontent.com/tianon/cgroupfs-mount/951c38ee8d0233045bdede20d85e1c0f8d312/cgroupfs-mount > cgroupfs-mount.sh
chmod +x cgroupfs-mount.sh
sudo bash ./cgroupfs-mount.sh
```

Your Amazon EC2 instance should now be ready for AWS IoT Greengrass.

7. To make sure that you have all required dependencies, download and run the Greengrass dependency checker from the AWS IoT Greengrass Samples repository on GitHub. These commands unzip and run the dependency checker script.

```bash
mkdir greengrass-dependency-checker-GGCv1.9.x
cd greengrass-dependency-checker-GGCv1.9.x
unzip greengrass-dependency-checker-GGCv1.9.x.zip
cd greengrass-dependency-checker-GGCv1.9.x
sudo ./check_ggc_dependencies | more
```

Important
This tutorial requires Python 2.7. The `check_ggc_dependencies` script might produce warnings about the missing optional Node.js and Java prerequisites. You can ignore these warnings.
Your Amazon EC2 instance configuration is complete. Continue to the section called “Module 2: Installing the AWS IoT Greengrass Core Software” (p. 76).

Setting Up Other Devices

If you’re new to AWS IoT Greengrass, we recommend that you use a Raspberry Pi or an Amazon EC2 instance as your core device, and follow the setup steps (p. 65) appropriate for your device. To use a different device or platform, follow the steps in this section. For information about supported device platforms, see Greengrass Core Platform Compatibility.

1. If your core device is an NVIDIA Jetson TX2, you must first flash the firmware with the JetPack 3.3 installer. If you’re configuring a different device, skip to step 2.

   **Note**
   The JetPack installer version that you use is based on your target CUDA Toolkit version. The following instructions use JetPack 3.3 and CUDA Toolkit 9.0 because the TensorFlow v1.10.1 and MXNet v1.2.1 binaries (that AWS IoT Greengrass provides for machine learning inference on a Jetson TX2) are compiled against this version of CUDA. For more information, see Perform Machine Learning Inference (p. 221).

   a. On a physical desktop that is running Ubuntu 16.04 or later, flash the firmware with the JetPack 3.3 installer, as described in Download and Install JetPack (3.3) in the NVIDIA documentation.

      Follow the instructions in the installer to install all the packages and dependencies on the Jetson board, which must be connected to the desktop with a Micro-B cable.

   b. Reboot your board in normal mode, and connect a display to the board.

      **Note**
      When you use SSH to connect to the Jetson board, use the default user name (nvidia) and the default password (nvidia).

2. Run the following commands to create user *ggc_user* and group *ggc_group*. The commands you run differ, depending on the distribution installed on your core device.

   - If your core device is running OpenWrt, run the following commands:

     ```bash
     opkg install shadow-useradd
     opkg install shadow-groupadd
     useradd --system ggc_user
     groupadd --system ggc_group
     ```

   - Otherwise, run the following commands:

     ```bash
     sudo adduser --system ggc_user
     sudo addgroup --system ggc_group
     ```

      **Note**
      If the *addgroup* command isn't available on your system, use the following command.

      ```bash
      sudo groupadd --system ggc_group
      ```

3. To make sure that you have all required dependencies, download and run the Greengrass dependency checker from the AWS IoT Greengrass Samples repository on GitHub. These commands unzip and run the dependency checker script.

    ```bash
    mkdir greengrass-dependency-checker-GGCv1.9.x
    cd greengrass-dependency-checker-GGCv1.9.x
    ```
unzip greengrass-dependency-checker-GGCv1.9.x.zip
cd greengrass-dependency-checker-GGCv1.9.x
sudo ./check_ggc_dependencies | more

**Note**
The `check_ggc_dependencies` script runs on AWS IoT Greengrass supported platforms and requires the following Linux system commands: `printf`, `uname`, `cat`, `ls`, `head`, `find`, `zcat`, `awk`, `sed`, `sysctl`, `wc`, `cut`, `sort`, `expr`, `grep`, `test`, `dirname`, `readlink`, `xargs`, `strings`, `uniq`.
For more information, see the dependency checker's Readme.

4. Install all required dependencies on your device, as indicated by the dependency checker output. For missing kernel-level dependencies, you might have to recompile your kernel. For mounting Linux control groups (`cgroups`), you can run the `cgroupfs-mount` script. This allows AWS IoT Greengrass to set the memory limit for Lambda functions. Cgroups are also required to run AWS IoT Greengrass in the default `containerization` (p. 181) mode.

If no errors appear in the output, AWS IoT Greengrass should be able to run successfully on your device.

**Important**
This tutorial requires Python 2.7. The `check_ggc_dependencies` script might produce warnings about the missing optional Node.js and Java prerequisites. You can ignore these warnings.

For the list of AWS IoT Greengrass requirements and dependencies, see Supported Platforms and Requirements (p. 11).

---

### Module 2: Installing the AWS IoT Greengrass Core Software

This module shows you how to install the AWS IoT Greengrass Core software on your chosen device. This tutorial provides instructions for setting up a Raspberry Pi, but you can use any supported device. You can download the AWS IoT Greengrass Core software from the AWS IoT Greengrass Core Software (p. 16) downloads. This procedure includes steps for configuring and starting the software on your device.

The AWS IoT Greengrass Core software provides the following functionality:

- Deployment and local execution of connectors and Lambda functions.
- Secure, encrypted storage of local secrets and controlled access by connectors and Lambda functions.
- MQTT messaging over the local network between devices, connectors, and Lambda functions using managed subscriptions.
- MQTT messaging between AWS IoT and devices, connectors, and Lambda functions using managed subscriptions.
- Secure connections between devices and the cloud using device authentication and authorization.
- Local shadow synchronization of devices. Shadows can be configured to sync with the cloud.
- Controlled access to local device and volume resources.
- Deployment of cloud-trained machine learning models for running local inference.
- Automatic IP address detection that enables devices to discover the Greengrass core device.
- Central deployment of new or updated group configuration. After the configuration data is downloaded, the core device is restarted automatically.
- Secure, over-the-air software updates of user-defined Lambda functions.
Configure AWS IoT Greengrass on AWS IoT

Before you begin, make sure that you have completed Module 1 (p. 65).
This module should take less than 30 minutes to complete.

**Topics**

- Configure AWS IoT Greengrass on AWS IoT (p. 77)
- Start AWS IoT Greengrass on the Core Device (p. 81)

### Configure AWS IoT Greengrass on AWS IoT

1. Sign in to the AWS Management Console on your computer and open the AWS IoT console. If this is your first time opening this console, choose **Get started**.
2. Choose **Greengrass**.
   
   **Note**
   If you don't see the **Greengrass** node in the navigation pane, change to an AWS Region that supports AWS IoT Greengrass. For the list of supported regions, see **AWS Regions and Endpoints** in the **AWS General Reference**.

3. On the **Welcome to AWS IoT Greengrass** page, choose **Create a Group**.

   An AWS IoT Greengrass **group** (p. 6) contains settings and other information about its components, such as devices, Lambda functions, and connectors. A group defines how its components can interact with each other.

   **Tip**
   For an example that uses the AWS IoT Greengrass API to create and deploy a group, see the **gg_group_setup** package from GitHub.
4. If prompted, on the **Greengrass needs your permission to access other services** dialog box, choose **Grant permission** to allow the console to create or configure the Greengrass service role for you. You must use a service role to authorize AWS IoT Greengrass to access other AWS services on your behalf. Otherwise, deployments fail.

![Greengrass needs your permission to access other services](image)

AWS IoT Greengrass works with other AWS services, such as AWS IoT and AWS Lambda. Greengrass needs your permission to access these services and read and write data on your behalf. [Learn more](#)

When you grant permission, Greengrass does the following:
- Creates a service role named Greengrass_ServiceRole, if one doesn’t exist, and attaches the AWSGreengrassResourceAccessRolePolicy managed policy to the role.
- Attaches the service role to your AWS account in the AWS Region that’s currently selected in the console.

This step is required only once in each AWS Region where you use Greengrass.

![Grant permission](image)

The AWS account you used to sign in must have permissions to create or manage the IAM role. For more information, see the section called “Greengrass Service Role” (p. 443).

5. On the **Set up your Greengrass group** page, choose **Use easy creation** to create a group and an AWS IoT Greengrass core (p. 24).

Each group requires a core, which is a device that manages local IoT processes. A core needs a certificate and keys that allow it to access AWS IoT and an AWS IoT policy that allows it to perform AWS IoT and AWS IoT Greengrass actions. When you choose the **Use easy creation** option, these security resources are created for you and the core is provisioned in the AWS IoT registry.

![Set up your Greengrass Group](image)

6. Enter a name for your group (for example, *MyFirstGroup*), and then choose **Next**.
7. Use the default name for the AWS IoT Greengrass core, and then choose Next.

8. On the Run a scripted easy Group creation page, choose Create Group and Core.
AWS IoT creates an AWS IoT Greengrass group with default security policies and configuration files for you to load onto your device.

9. Download your core's security resources and configuration file. On the confirmation page, under Download and store your Core's security resources, choose Download these resources as a tar.gz. The name of your downloaded tar.gz file starts with a 10-digit hash that's also used for the certificate and key file names.

**Important**
Download the security resources before you choose Finish.

10. After you download the security resources, choose Finish.

The group configuration page is displayed in the console:
11. Download the AWS IoT Greengrass Core software (p. 16) installation package. Choose the CPU architecture and distribution (and operating system, if necessary) that best describe your core device. For example:

- For Raspberry Pi Model B or B+, download the Armv7l for Raspbian package.
- For an Amazon EC2 instance, download the x86_64 for Linux package.
- For NVIDIA Jetson TX2, download the Armv8 (AArch64) for Ubuntu package.
- For Intel Atom, download the x86_64 for Linux package.

Start AWS IoT Greengrass on the Core Device

**Note**
This tutorial provides instructions for starting AWS IoT Greengrass on your Raspberry Pi, but you can use any supported device.

In a previous step (p. 80), you downloaded two files to your computer:

- `greengrass-OS-architecture-1.9.4.tar.gz`. This compressed file contains the AWS IoT Greengrass Core software that runs on the core device.
- `hash-setup.tar.gz` (for example, `c6973960cc-setup.tar.gz`). This compressed file contains security certificates that enable secure communications between AWS IoT and the `config.json` file that contains configuration information specific to your AWS IoT Greengrass core and the AWS IoT endpoint.

1. If you don’t know the IP address of your AWS IoT Greengrass core device, open a terminal on the AWS IoT Greengrass core device and run the following command:

   **Note**
   This command might not return the correct IP address for some devices. Consult the documentation for your device to retrieve your device IP address.

   ```bash
   hostname -I
   ```

2. Transfer the two compressed files from your computer to the AWS IoT Greengrass core device. Choose your operating system for steps that show how to transfer files to your Raspberry Pi device. The file transfer steps vary, depending on device or EC2 instance.
Note
For a Raspberry Pi, the default user name is **pi** and the default password is **raspberry**. For an NVIDIA Jetson TX2, the default user name is **nvidia** and the default password is **nvidia**.

Windows

To transfer the compressed files from your computer to a Raspberry Pi AWS IoT Greengrass core device, use a tool such as WinSCP or the PuTTY `pscp` command. To use the `pscp` command, open a Command Prompt window on your computer and run the following:

```
cd path-to-downloaded-files
pscp -pw Pi-password greengrass-OS-architecture-1.9.4.tar.gz pi@IP-address:/home/pi
pscp -pw Pi-password hash-setup.tar.gz pi@IP-address:/home/pi
```

Note
The version number in this command must match the version of your AWS IoT Greengrass Core software package.

macOS

To transfer the compressed files from your Mac to a Raspberry Pi AWS IoT Greengrass core device, open a Terminal window on your computer and run the following commands. The `path-to-downloaded-files` is typically `~/Downloads`.

```
cd path-to-downloaded-files
scp greengrass-OS-architecture-1.9.4.tar.gz pi@IP-address:/home/pi
scp hash-setup.tar.gz pi@IP-address:/home/pi
```

Note
The version number in this command must match the version of your AWS IoT Greengrass Core software package.

UNIX-like system

To transfer the compressed files from your computer to a Raspberry Pi AWS IoT Greengrass core device, open a terminal window on your computer and run the following commands:

```
cd path-to-downloaded-files
scp greengrass-OS-architecture-1.9.4.tar.gz pi@IP-address:/home/pi
scp hash-setup.tar.gz pi@IP-address:/home/pi
```

Note
The version number in this command must match the version of your AWS IoT Greengrass Core software package.

Raspberry Pi web browser

If you used the Raspberry Pi's web browser to download the compressed files, the files should be in the Pi's `~/Downloads` folder (for example, `/home/pi/Downloads`). Otherwise, the compressed files should be in the Pi's `~` folder (for example, `/home/pi`).

3. Open a terminal on the AWS IoT Greengrass core device and navigate to the folder that contains the compressed files (for example, `cd /home/pi`).
Start AWS IoT Greengrass on the Core Device

4. Decompress the AWS IoT Greengrass Core software and the security resources.

   • The first command creates the /greengrass directory in the root folder of the AWS IoT Greengrass core device (through the -C / argument).
   • The second command copies the certificates into the /greengrass/certs folder and the config.json (p. 24) file into the /greengrass/config folder (through the -C /greengrass argument).

   ```
sudo tar -xzvf greengrass-OS-architecture-1.9.4.tar.gz -C /
sudo tar -xzvf hash-setup.tar.gz -C /greengrass
```

   **Note**
   
The version number in this command must match the version of your AWS IoT Greengrass Core software package.

5. Review Server Authentication in the AWS IoT Developer Guide and choose the appropriate root CA certificate. We recommend that you use Amazon Trust Services (ATS) endpoints and ATS root CA certificates. Certificates enable your device to communicate with AWS IoT using the MQTT messaging protocol over TLS.

   Make sure that the AWS IoT Greengrass core device is connected to the internet, and then download the root CA certificate to your core device.

   **Important**
   
   Your root CA certificate type must match your endpoint. Use an ATS root CA certificate with an ATS endpoint (preferred) or a Verisign root CA certificate with a legacy endpoint. Only some AWS Regions support legacy endpoints. For more information, see the section called “Endpoints Must Match the Certificate Type” (p. 45).

   For ATS endpoints (preferred), download the appropriate ATS root CA certificate. The following example downloads AmazonRootCA1.pem. The `wget -O` parameter is the capital letter O.

   ```
cd /greengrass/certs/
sudo wget -O root.ca.pem https://www.amazontrust.com/repository/AmazonRootCA1.pem
```

   **Note**
   
   For legacy endpoints, download a Verisign root CA certificate. Although legacy endpoints are acceptable for the purposes of this tutorial, we recommend that you create an ATS endpoint and download an ATS root CA certificate.

   ```
cd /greengrass/certs/
```

   You can run the following command to confirm that the root.ca.pem file is not empty:

   ```
cat root.ca.pem
```

   If the root.ca.pem file is empty, check the wget URL and try again.

6. Start AWS IoT Greengrass on your core device.

   ```
cd /greengrass/ggc/core/
```
You should see a `Greengrass successfully started` message. Make a note of the PID.

**Note**
To set up your core device to start AWS IoT Greengrass on system boot, see the section called “Start Greengrass on System Boot” (p. 62).

You can run the following command to confirm that the AWS IoT Greengrass Core software (daemon) is functioning. Replace `PID-number` with your PID:

```
ps aux | grep PID-number
```

You should see an entry for the PID with a path to the running Greengrass daemon (for example, `/greengrass/ggc/packages/1.9.4/bin/daemon`). If you run into issues starting AWS IoT Greengrass, see Troubleshooting (p. 516).

Module 3 (Part 1): Lambda Functions on AWS IoT Greengrass

This module shows you how to configure a Lambda function and deploy it to your AWS IoT Greengrass core device. It contains information about MQTT messaging, subscriptions, deployments on AWS IoT Greengrass, and Lambda function configurations.

Part 1 of this module shows you how to deploy a Lambda function on the AWS IoT Greengrass core that sends Hello World messages to the AWS IoT Greengrass cloud. Part 2 (p. 97) covers the differences between on-demand and long-lived Lambda functions running on the AWS IoT Greengrass core.

Before you begin, make sure that you have completed Module 1 (p. 65) and Module 2 (p. 76) and have a running AWS IoT Greengrass core device.

Module 3 (Part 1) (p. 84) and Module 3 (Part 2) (p. 97) should take approximately 30 minutes each to complete.

**Topics**
- Create and Package a Lambda Function (p. 84)
- Configure the Lambda Function for AWS IoT Greengrass (p. 88)
- Deploy Cloud Configurations to an AWS IoT Greengrass Core Device (p. 93)
- Verify the Lambda Function Is Running on the Device (p. 94)

Create and Package a Lambda Function

To run on an AWS IoT Greengrass core, a Python Lambda function requires the AWS IoT Greengrass Core SDK for Python. For this tutorial, you include the `greengrasssdk` folder in the Lambda function deployment package.

**Note**
If you're running Python Lambda functions, you can also use `pip` to install the AWS IoT Greengrass Core SDK for Python on the core device. Then you can deploy your functions without including the SDK in the Lambda function deployment package. For more information, see `greengrasssdk`.

In this section, you:
• Download the AWS IoT Greengrass Core SDK for Python to your computer (not the AWS IoT Greengrass core device). For this tutorial, you download the SDK on GitHub from the AWS IoT Greengrass Core SDK (p. 22) downloads page.

• Get the Python Lambda function (named greengrassHelloWorld.py) from the downloaded SDK.

• Create a Lambda function deployment package named hello_world_python_lambda.zip that contains greengrassHelloWorld.py and the greengrasssdk folder.

• Use the Lambda console to upload the hello_world_python_lambda.zip package.

• Use the AWS IoT console to transfer the package to the AWS IoT Greengrass core device (during group deployment).

1. Download the AWS IoT Greengrass Core SDK for Python from the AWS IoT Greengrass Core SDK (p. 22) downloads page.

2. Unzip the downloaded package to get the Lambda function code and the SDK.

   The Lambda function in this module uses:

   • The greengrassHelloWorld.py file in examples\HelloWorld. This is your Lambda function code.

   • The greengrasssdk folder. This is the SDK.

   The following code is from greengrassHelloWorld.py. (To save space, all code comments have been removed.) Note that every five seconds, the function publishes one of two possible messages to the hello/world topic.

   ```python
   import greengrasssdk
   import platform
   from threading import Timer
   import time

   client = greengrasssdk.client('iot-data')
   my_platform = platform.platform()

   def greengrass_hello_world_run():
       if not my_platform:
           client.publish(topic='hello/world', payload='Hello world! Sent from Greengrass Core. ')
       else:
           client.publish(topic='hello/world', payload='Hello world! Sent from Greengrass Core running on platform: {}'.format(my_platform))

   Timer(5, greengrass_hello_world_run).start()

   greengrass_hello_world_run()

   def function_handler(event, context):
       return
   ```

3. Copy greengrasssdk to the HelloWorld folder that contains greengrassHelloWorld.py.

4. To create the Lambda function deployment package, save the greengrassHelloWorld.py file and the greengrasssdk folder to a compressed .zip file named hello_world_python_lambda.zip. The .py file and SDK folder must be in the root of the directory.
For UNIX-like systems (including the Mac terminal), you can use the following command to package the file and folder:

```bash
sudo zip -r hello_world_python_lambda.zip greengrassdk greengrassHelloWorld.py
```

**Note**
Depending on your distribution, you might need to install `zip` first (for example, by running `sudo apt-get install zip`). The installation command for your distribution might be different.

Now you're ready to create your Lambda function and upload the deployment package.

5. Open the Lambda console and choose **Create function**.
6. Choose **Author from scratch**.
7. Name your function **Greengrass_HelloWorld**, and set the remaining fields as follows:
   - For **Runtime**, choose **Python 2.7**.
   - For **Permissions**, keep the default setting. This creates an execution role that grants basic Lambda permissions. This role isn't used by AWS IoT Greengrass.

Choose **Create function**.

8. Upload your Lambda function deployment package:
   a. On the **Configuration** tab, under **Function code**, set the following fields:
      - For **Code entry type**, choose **Upload a .zip file**.
      - For **Runtime**, choose **Python 2.7**.
      - For **Handler**, enter `greengrassHelloWorld.function_handler`
   b. Choose **Upload**, and then choose `hello_world_python_lambda.zip`. (The size of your `hello_world_python_lambda.zip` file might be different from what's shown here.)
Create and Package a Lambda Function

c. Choose **Save**.

**Note**
The **Test** button on the AWS Lambda console doesn't work with this function. The AWS IoT Greengrass Core SDK doesn't contain modules that are required to run your Greengrass Lambda functions independently in the AWS Lambda console. These modules (for example, `greengrass_common`) are supplied to the functions after they are deployed to your Greengrass core.

**Tip**
To see your uploaded code, from **Code entry type**, choose **Edit code inline**.

9. Publish the Lambda function:

a. From **Actions**, choose **Publish new version**.

   ![Function code](image)

b. For **Version description**, enter **First version**, and then choose **Publish**.

   ![Publish new version](image)

10. Create an **alias** for the Lambda function version:

    **Note**
    Greengrass groups can reference a Lambda function by alias (recommended) or by version. Using an alias makes it easier to manage code updates because you don't have to change
your subscription table or group definition when the function code is updated. Instead, you just point the alias to the new function version.

a. From **Actions**, choose **Create alias**.

b. Name the alias **GG_HelloWorld**, set the version to 1 (which corresponds to the version that you just published), and then choose **Create**.

**Note**
AWS IoT Greengrass doesn't support Lambda aliases for $LATEST versions.

---

Configure the Lambda Function for AWS IoT Greengrass

You are now ready to configure your Lambda function for AWS IoT Greengrass. In this step, you also configure a subscription that allows the Lambda function to communicate with AWS IoT and configure local logging for the Greengrass group.

1. In the AWS IoT console, under **Greengrass**, choose **Groups**, and then choose the group that you created in Module 2 (p. 76).
2. On the group configuration page, choose **Lambdas**, and then choose **Add Lambda**.

3. Choose **Use existing Lambda**.

4. Search for the name of the Lambda you created in the previous step (**Greengrass_HelloWorld**, not the alias name), select it, and then choose **Next**:

5. For the version, choose **Alias: GG_HelloWorld**, and then choose **Finish**. You should see the **Greengrass_HelloWorld** Lambda function in your group, using the **GG_HelloWorld** alias.

6. Choose the ellipsis (...), and then choose **Edit Configuration**:
7. On the **Group-specific Lambda configuration** page, make the following changes:

- Set **Timeout** to 25 seconds. This Lambda function sleeps for 20 seconds before each invocation.
- For **Lambda lifecycle**, choose **Make this function long-lived and keep it running indefinitely**.

### Note

A *long-lived* Lambda function starts automatically after AWS IoT Greengrass starts and keeps running in its own container (or sandbox). This is in contrast to an *on-demand* Lambda function, which starts when invoked and stops when there are no tasks left to execute. For more information, see the section called “Lifecycle Configuration” (p. 187).

8. Keep the default values for all other fields, such as **Run as**, **Containerization**, and **Input payload data type**, and choose **Update** to save your changes. For information about Lambda function properties, see the section called “Controlling Greengrass Lambda Function Execution” (p. 178).

Next, create a subscription that allows the Lambda to send MQTT messages to AWS IoT.

A Greengrass Lambda function can exchange MQTT messages with:

- **Devices** (p. 8) in the Greengrass group.
- **Connectors** (p. 283) in the group.
- Other Lambda functions in the group.
- **AWS IoT**.
- The local shadow service. For more information, see the section called “Module 5: Interacting with Device Shadows” (p. 122).
The group uses subscriptions to control how these entities can communicate with each other. Subscriptions provide predictable interactions and a layer of security.

A subscription consists of a source, target, and topic. The source is the originator of the message. The target is the destination of the message. The topic allows you to filter the data that is sent from the source to the target. The source or target can be a Greengrass device, Lambda function, connector, device shadow, or AWS IoT.

**Note**

A subscription is directed in the sense that messages flow in a specific direction: from the source to the target. To allow two-way communication, you must set up two subscriptions.

The Greengrass_HelloWorld Lambda function sends messages only to the hello/world topic in AWS IoT, as shown in the following `greengrassHelloWorld.py` code snippet:

```python
def greengrass_hello_world_run():
    if not my_platform:
        client.publish(topic='hello/world', payload='Hello world! Sent from Greengrass Core.
    else:
        client.publish(topic='hello/world', payload='Hello world! Sent from Greengrass Core running on platform: {}'.format(my_platform))
    Timer(5, greengrass_hello_world_run).start()
```

Because the Greengrass_HelloWorld Lambda function sends messages only to the hello/world topic in AWS IoT, you only need to create one subscription from the Lambda function to AWS IoT, as shown next.

9. On the group configuration page, choose **Subscriptions**, and then choose **Add your first Subscription**.

10. In **Select a source**, choose **Select**. Then, on the **Lambdas** tab, choose **Greengrass_HelloWorld** as the source.
11. For **Select a target**, choose **Select**. Then, on the **Service** tab, choose **IoT Cloud**, and then choose **Next**.

12. For **Topic filter**, enter **hello/world**, and then choose **Next**.

14. Configure the group's logging settings. For this tutorial, you configure AWS IoT Greengrass system components and user-defined Lambda functions to write logs to the file system of the core device. For more information, see Monitoring (p. 457).
   a. On the group configuration page, choose Settings.
   b. For Local logs configuration, choose Edit.
   c. On the Configure Group logging page, choose Add another log type.
   d. For event source, choose User Lambdas and Greengrass system, and then choose Update.
   e. Keep the default values for logging level and disk space limit, and then choose Save.

Deploy Cloud Configurations to an AWS IoT Greengrass Core Device

1. Make sure that your AWS IoT Greengrass core device is connected to the internet. (For example, see if you can successfully navigate to a webpage.)

2. Make sure that the AWS IoT Greengrass daemon is running on your core device. Run the following commands in your core device terminal.
   a. To check whether the daemon is running:

   ```
   ps aux | grep -E 'greengrass.*daemon'
   ```
   
   If the output contains a root entry for /greengrass/ggc/packages/1.9.4/bin/daemon, then the daemon is running.
   
   b. To start the daemon:

   ```
   cd /greengrass/ggc/core/
   sudo ./greengrassd start
   ```
   
   Now you're ready to deploy the Lambda function and subscription configurations to your AWS IoT Greengrass core device.

3. In the AWS IoT console, on the group configuration page, from Actions, choose Deploy.
4. On the **Configure how devices discover your core** page, choose **Automatic detection**. This enables devices to automatically acquire connectivity information for the core, such as IP address, DNS, and port number. Automatic detection is recommended, but AWS IoT Greengrass also supports manually specified endpoints. You’re only prompted for the discovery method the first time that the group is deployed.

The first deployment might take a few minutes. When the deployment is complete, you should see **Successfully completed** in the **Status** column on the **Deployments** page:

**Note**
The deployment status is also displayed below the group's name on the page header.

For troubleshooting help, see *Troubleshooting* (p. 516).

**Verify the Lambda Function Is Running on the Device**

1. From the navigation pane of the AWS IoT console, choose **Test**.
2. Choose **Subscribe to topic**, and configure the following fields:

   - For **Subscription topic**, enter *hello/world*. (Don't choose **Subscribe to topic** yet.)
   - For **Quality of Service**, choose 0.
   - For **MQTT payload display**, choose **Display payloads as strings**.
3. Choose **Subscribe to topic**.

Assuming the Lambda function is running on your device, it publishes messages similar to the following to the `hello/world` topic:

Although the Lambda function continues to send MQTT messages to the `hello/world` topic, don't stop the AWS IoT Greengrass daemon. The remaining modules are written with the assumption that it's running.
You can delete the function and subscription from the group:

- From the **Lambdas** page, choose the ellipsis (…), and then choose **Remove function**.
- From the **Subscriptions** page, choose the ellipsis (…), and then choose **Delete**.

The function and subscription are removed from the core during the next group deployment.

---

**Module 3 (Part 2): Lambda Functions on AWS IoT Greengrass**

This module shows you how to configure a Lambda function and deploy it to your AWS IoT Greengrass core device. It contains information about MQTT messaging, subscriptions, deployments on AWS IoT Greengrass, and Lambda function configurations.

**Part 1** (p. 84) of this module described how to deploy a Lambda function on a AWS IoT Greengrass core that sends Hello World messages to AWS IoT. This part explores the differences between on-demand and long-lived Lambda functions running on the AWS IoT Greengrass core.

Before you begin, make sure you have completed **Module 1** (p. 65), **Module 2** (p. 76), and **Module 3 (Part 1)** (p. 84).

This module should take approximately 30 minutes to complete.

**Topics**

- Create and Package the Lambda Function (p. 97)
- Configure Long-Lived Lambda Functions for AWS IoT Greengrass (p. 99)
- Test Long-Lived Lambda Functions (p. 100)
- Test On-Demand Lambda Functions (p. 105)

**Create and Package the Lambda Function**

1. **Download the Lambda function code to your computer (not the Greengrass core device):**
   
   a. In a web browser, open the `greengrassHelloWorldCounter.py` file on GitHub.
   
   b. Choose **Raw** to open the unformatted version of the file.

   ![Raw Button](image)

   c. **Use Ctrl + S (or Command + S for the Mac) to save a copy of the `greengrassHelloWorldCounter.py` file. Save the file to a folder that contains the `greengrasssdk` folder.**

   **Note**
   
   For UNIX-like systems, you can run the following Terminal command to download the `greengrassHelloWorldCounter.py` file:

   ```
   wget https://raw.githubusercontent.com/aws-samples/aws-greengrass-samples/master/hello-world-counter-python/greengrassHelloWorldCounter.py
   ```

   97
2. Package the `greengrassHelloWorldCounter.py` file with the SDK into a `.zip` file, as described in Module 3 (Part 1) (p. 84). Name the package `hello_world_counter_python_lambda.zip`.

3. In the Lambda console, create a Python 2.7 function named `Greengrass_HelloWorld_Counter`, as described in Module 3 (Part 1) (p. 84). You can use the existing role.

4. Upload your Lambda function deployment package:
   a. On the Configuration tab, under Function code, set the following fields:
      - For Code entry type, choose Upload a `.zip` file.
      - For Runtime, choose Python 2.7.
      - For Handler, enter `greengrassHelloWorldCounter.function_handler`
   b. Choose Upload, and then choose `hello_world_counter_python_lambda.zip`.
   c. At the top of the page, choose Save.

   **Note**
   The Test button on the AWS Lambda console doesn't work with this function. The AWS IoT Greengrass Core SDK doesn't contain modules that are required to run your Greengrass Lambda functions independently in the AWS Lambda console. These modules (for example, `greengrass_common`) are supplied to the functions after they are deployed to your Greengrass core.

5. Publish the first version of the function:
   a. From Actions, choose Publish new version. For Version description, enter First version.
   b. Choose Publish.

6. Create an alias for the function version:
   a. From the Actions menu, choose Create alias, and set the following values:
      - For Name, enter `GG_HW_Counter`.
      - For Version, choose 1.
   b. Choose Create.
Configure Long-Lived Lambda Functions for AWS IoT Greengrass

You are now ready to configure your Lambda function for AWS IoT Greengrass.

1. In the AWS IoT console, under Greengrass, choose Groups, and then choose the group that you created in Module 2 (p. 76).
2. On the group configuration page, choose Lambdas, and then choose Add Lambda.
3. On the Add a Lambda to your Greengrass Group page, choose Use existing Lambda.
4. On the Use existing Lambda page, choose Greengrass_HelloWorld_Counter, and then choose Next.
5. On the Select a Lambda version page, choose Alias: GG_HW_Counter, and then choose Finish.
6. On the Lambdas page, from the ... menu, choose Edit Configuration.
7. On the configuration page, edit the following properties:
   - Set **Timeout** to 25 seconds. This Lambda function sleeps for 20 seconds before each invocation.
   - For **Lambda lifecycle**, choose **Make this function long-lived and keep it running indefinitely**.
   - Accept the default values for all other fields, such as **Run as** and **Containerization**.

8. Choose **Update**.

**Test Long-Lived Lambda Functions**

A long-lived (p. 187) Lambda function starts automatically when the AWS IoT Greengrass core starts and runs in a single container (or sandbox). Any variables or preprocessing that are defined outside of the function handler are retained for every invocation of the function handler. Multiple invocations of the function handler are queued until earlier invocations have been executed.

The following code is from `greengrassHelloWorldCounter.py`. (Code comments are removed for brevity.) This Lambda function is similar to the `greengrassHelloWorld.py` function from Part 1 of this module, but it defines a `my_counter` variable outside of the function handler.
import greengrasssdk
import platform
import time
import json

client = greengrasssdk.client('iot-data')
my_platform = platform.platform()
my_counter = 0

def function_handler(event, context):
    global my_counter
    my_counter = my_counter + 1
    if not my_platform:
        client.publish(
            topic='hello/world/counter',
            payload=json.dumps({'message': 'Hello world! Sent from Greengrass Core. Invocation Count: {}'.format(my_counter)})
        )
    else:
        client.publish(
            topic='hello/world/counter',
            payload=json.dumps({'message': 'Hello world! Sent from Greengrass Core running on platform: {}. Invocation Count: {}' .format(my_platform, my_counter)})
        )
    time.sleep(20)
    return

In this step, you create subscriptions that allow the Lambda function and AWS IoT to exchange MQTT messages. Then you deploy the group and test the function.

1. On the group configuration page, choose Subscriptions, and then choose Add Subscription.
2. Under Select a source, choose the Lambdas tab, and then choose Greengrass_HelloWorld_Counter.
3. Under Select a target, choose the Services tab, choose IoT Cloud, and then choose Next.

![CREATE A SUBSCRIPTION]

Select your source and target

A Subscription consists of a source, target, and topic. The source is the originator of the message. The target is the destination of the message. The first step is selecting your source and target.

Select a source

Greengrass_HelloWorld_Counter  LAMBDA

Select a target

IoT Cloud  SERVICE

4. For Topic filter, enter hello/world/counter. Choose Next, and then choose Finish.
This single subscription goes in one direction only: from the `Greengrass_HelloWorld_Counter` Lambda function to AWS IoT. To invoke (or trigger) this Lambda function from the cloud, you must create a subscription in the opposite direction.

5. Follow steps 1 - 4 to add another subscription that uses the following values. This subscription allows the Lambda function to receive messages from AWS IoT. This subscription is exercised later when you send a test message to the function from the console.

   • For the source, choose `Services`, and then choose `IoT Cloud`.
   • For the target, choose `Lambdas`, and then choose `Greengrass_HelloWorld_Counter`.
   • For the topic filter, enter `hello/world/counter/trigger`.

The `/trigger` extension is used in this topic filter because you created two subscriptions and don't want them to interfere with each other.

6. Make sure that the AWS IoT Greengrass daemon is running, as described in Deploy Cloud Configurations to a Core Device (p. 93).

7. On the group configuration page, from `Actions`, choose `Deploy`.

Your Subscription is complete and your objects are connected in this Group. You can now save, and then deploy your new Group definition to have this change take effect.
This deploys the group configuration to your AWS IoT Greengrass core device. For troubleshooting help, see Troubleshooting (p. 516).

8. After your deployment is complete, return to the AWS IoT console home page and choose Test.

9. Configure the following fields:

   - For **Subscription topic**, enter `hello/world/counter`.
   - For **Quality of Service**, choose 0.
   - For **MQTT payload display**, choose **Display payloads as strings**.

10. Choose **Subscribe to topic**.

    Unlike Part 1 (p. 84) of this module, you shouldn't see any messages after you subscribe to `hello/world/counter`. This is because the `greengrassHelloWorldCounter.py` code that publishes to the `hello/world/counter` topic is inside the function handler, which runs only when the function is invoked.

    In this module, you configured the **Greengrass_HelloWorld_Counter** Lambda function to be invoked when it receives an MQTT message on the `hello/world/counter/trigger` topic. You can see this by examining the related subscriptions:
The **Greengrass_HelloWorld_Counter** to **IoT Cloud** subscription allows the function to send messages to AWS IoT on the **hello/world/counter** topic. The **IoT Cloud** to **Greengrass_HelloWorld_Counter** subscription allows AWS IoT to send messages to the function on the **hello/world/counter/trigger** topic.

**Note**

**Greengrass_HelloWorld_Counter** ignores the content of received messages. It just runs the code in **function_handler**, which sends a message to the **hello/world/counter** topic. To review this code, see the `greengrassHelloWorldCounter.py` (p. 100) code listing.

11. To test the long-lived lifecycle, invoke the Lambda function by publishing a message to the **hello/world/counter/trigger** topic. You can use the default message.

Every time a message is published to the **hello/world/counter/trigger** topic, the **my_counter** variable is incremented. This invocation count is shown in the messages sent from the Lambda function. Because the function handler includes a 20-second sleep cycle (`time.sleep(20)`), repeatedly triggering the handler queues up responses from the AWS IoT Greengrass core.
An on-demand (p. 187) Lambda function is similar in functionality to a cloud-based AWS Lambda function. Multiple invocations of an on-demand Lambda function can run in parallel. An invocation of the Lambda function creates a separate container to process invocations or reuses an existing container, if resources permit. Any variables or preprocessing that are defined outside of the function handler are not retained when containers are created.

1. On the group configuration page, choose **Lambdas**.
2. For the **Greengrass_HelloWorld_Counter** Lambda function, choose **Edit Configuration**.
3. Under **Lambda lifecycle**, choose **On-demand function**, and then choose **Update**.
4. On the group configuration page, from Actions, choose Deploy.

This deploys the group configuration to your AWS IoT Greengrass core device. For troubleshooting help, see Troubleshooting (p. 516).

5. After your deployment is complete, return to the AWS IoT console home page and choose Test.

6. Configure the following fields:
   - For Subscription topic, enter hello/world/counter.
   - For Quality of Service, choose 0.
   - For MQTT payload display, choose Display payloads as strings.
7. Choose **Subscribe to topic**.

   **Note**
   You should not see any messages after you subscribe.

8. To test the on-demand lifecycle, invoke the function by publishing a message to the **hello/world/counter/trigger** topic. You can use the default message.

   a. Choose **Publish to topic** three times quickly, within five seconds of each press of the button.

Each publish invokes the function handler and creates a container for each invocation. The invocation count is not incremented for the three times you triggered the function because each on-demand Lambda function has its own container/sandbox.
b. After approximately 30 seconds, choose **Publish to topic**. The invocation count should be incremented to 2. This shows that a container created from an earlier invocation is being reused, and that preprocessing variables outside of the function handler were stored.
You should now understand the two types of Lambda functions that can run on the AWS IoT Greengrass core. The next module, Module 4 (p. 109), shows you how devices can interact in an AWS IoT Greengrass group.

Module 4: Interacting with Devices in an AWS IoT Greengrass Group

This module shows you how AWS IoT devices can connect to and communicate with an AWS IoT Greengrass core device. AWS IoT devices that connect to an AWS IoT Greengrass core are part of an AWS IoT Greengrass group and can participate in the AWS IoT Greengrass programming paradigm. In this module, one Greengrass device sends a Hello World message to another device in the Greengrass group.

Before you begin, make sure that you have completed Module 1 (p. 65), Module 2 (p. 76), Module 3 (Part 1) (p. 84), and Module 3 (Part 2) (p. 97). You do not need other components or devices.

This module should take less than 30 minutes to complete.

Topics
- Create AWS IoT Devices in an AWS IoT Greengrass Group (p. 110)
- Configure Subscriptions (p. 113)
- Install the AWS IoT Device SDK for Python (p. 114)
- Test Communications (p. 119)
Create AWS IoT Devices in an AWS IoT Greengrass Group

1. In the AWS IoT console, choose **Greengrass**, choose **Groups**, and then choose your group.
2. On the group configuration page, choose **Devices**, and then choose **Add your first Device**.

3. Choose **Create New Device**.

4. Register this device as **HelloWorld_Publisher**, and then choose **Next**.
5. For 1-Click, choose **Use Defaults**. This option generates a device certificate with attached AWS IoT policy and public and private key.

6. Create a folder on your computer. Download the certificate and keys for your device into the folder.
7. Decompress the `hash-setup.tar.gz` file. For example, run the following command:

```
tar -xzf hash-setup.tar.gz
```

**Note**

On Windows, you can decompress `.tar.gz` files using a tool such as 7-Zip or WinZip.

8. Choose **Add Device** and repeat steps 3 - 7 to add a new device to the group.

Name this device **HelloWorld_Subscriber**. Download the certificates and keys for the device to your computer. Save and decompress them in the same folder that you created for **HelloWorld_Publisher**.

Again, make a note of the common `hash` component in the file names for the **HelloWorld_Subscriber** device.

You should now have two devices in your AWS IoT Greengrass group:
9. Review Server Authentication in the AWS IoT Developer Guide and choose the appropriate root CA certificate. We recommend that you use Amazon Trust Services (ATS) endpoints and ATS root CA certificates. Your root CA certificate type must match your endpoint. Use an ATS root CA certificate with an ATS endpoint (preferred) or a Verisign root CA certificate with a legacy endpoint. Only some AWS Regions support legacy endpoints. For more information, see the section called “Endpoints Must Match the Certificate Type” (p. 45).

Save the root CA certificate as root-ca-cert.pem in the same folder as the certificates and keys for both devices. All these files should be in one folder on your computer (not on the AWS IoT Greengrass core device).

- For ATS endpoints (preferred), download the appropriate ATS root CA certificate, such as Amazon Root CA 1.
- For legacy endpoints, download a Verisign root CA certificate. Although legacy endpoints are acceptable for the purposes of this tutorial, we recommend that you create an ATS endpoint and download an ATS root CA certificate.

**Note**
If you’re using a web browser on the Mac and you see This certificate is already installed as a certificate authority, open a Terminal window and download the certificate into the folder that contains the HelloWorld_Publisher and HelloWorld_Subscriber device certificates and keys. For example, if you’re using an ATS endpoint, you can run the following command to download the Amazon Root CA 1 certificate.

```
cd path-to-folder-containing-device-certificates
curl -o ./root-ca-cert.pem https://www.amazontrust.com/repository/AmazonRootCA1.pem
```

Run `cat root-ca-cert.pem` to ensure that the file is not empty. If the file is empty, check the URL and try the `curl` command again.

### Configure Subscriptions

In this step, you enable the HelloWorld_Publisher device to send MQTT messages to the HelloWorld_Subscriber device.

1. On the group configuration page, choose **Subscriptions**, and then choose **Add Subscription**.
2. Configure the subscription.

   - Under **Select a source**, choose **Devices**, and then choose **HelloWorld_Publisher**.
   - Under **Select a target**, choose **Devices**, and then choose **HelloWorld_Subscriber**.
   - Choose **Next**.
3. For **Topic filter**, enter `hello/world/pubsub`, choose **Next**, and then choose **Finish**.

   **Note**
   You can delete subscriptions from the previous modules. On the group’s **Subscriptions** page, choose the ellipsis (…) associated with a subscription, and then choose **Delete**.

4. Make sure that the AWS IoT Greengrass daemon is running, as described in **Deploy Cloud Configurations to a Core Device** (p. 93).

5. On the group configuration page, from **Actions**, choose **Deploy**.

   This deploys the group configuration to your AWS IoT Greengrass core device. For troubleshooting help, see **Troubleshooting** (p. 516).

The deployment status is displayed below the group name on the page header. To see deployment details, choose **Deployments**.

**Install the AWS IoT Device SDK for Python**

AWS IoT devices can use the AWS IoT Device SDK for Python to communicate with AWS IoT and AWS IoT Greengrass core devices (using the Python programming language). For more information, including requirements, see the AWS IoT Device SDK for Python SDK Readme on GitHub.

1. To install the SDK onto your computer, with all required components, choose your operating system:

   - **Windows**
     1. Open an **elevated command prompt** and run the following command:

        ```bash
        python --version
        ```
If no version information is returned or if the version number is less than 2.7 for Python 2 or less than 3.3 for Python 3, follow the instructions in Downloading Python to install Python 2.7+ or Python 3.3+. For more information, see Using Python on Windows.

2. Download the AWS IoT Device SDK for Python as a zip file and extract it to an appropriate location on your computer.

Make a note of the file path to the extracted aws-iot-device-sdk-python-master folder that contains the setup.py file. In the next step, this file path is indicated by `path-to-SDK-folder`.

3. From the elevated command prompt, run the following:

   ```shell
   cd path-to-SDK-folder
   python setup.py install
   ```

macOS

1. Open a Terminal window and run the following command:

   ```shell
   python --version
   ```

If no version information is returned or if the version number is less that 2.7 for Python 2 or less than 3.3 for Python 3, follow the instructions in Downloading Python to install Python 2.7+ or Python 3.3+. For more information, see Using Python on a Macintosh.

2. In the Terminal window, run the following commands to determine the OpenSSL version:

   ```python
   python
   >>>import ssl
   >>>print ssl.OPENSSL_VERSION
   ```

Make a note of the OpenSSL version value.

**Note**

If you're running Python 3, use `print(ssl.OPENSSL_VERSION)`.

To close the Python shell, run the following command:

```python
>>>exit()
```
AWS IoT Greengrass Developer Guide
Install the AWS IoT Device SDK for Python

Using pyenv

1. See Python Releases for Mac OS X (or similar) to determine the latest stable Python version. In the following example, this value is indicated by `latest-Python-version`.

2. From the Terminal window, run the following commands:

   ```bash
   pyenv install latest-Python-version
   pyenv global latest-Python-version
   ```

   For example, if the latest version for Python 2 is 2.7.14, then these commands are:

   ```bash
   pyenv install 2.7.14
   pyenv global 2.7.14
   ```

3. Close and then reopen the Terminal window and then run the following commands:

   ```python
   python
   >>> import ssl
   >>> print ssl.OPENSSL_VERSION
   ```

   The OpenSSL version should be at least 1.0.1. If the version is less than 1.0.1, then the update failed. Check the Python version value used in the `pyenv install` and `pyenv global` commands and try again.

4. Run the following command to exit the Python shell:

   ```python
   exit()
   ```

Not using pyenv

1. From a Terminal window, run the following command to determine if `brew` is installed:

   ```bash
   which brew
   ```

   If a file path is not returned, install `brew` as follows:

   ```bash
   /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
   ```

   **Note**
   Follow the installation prompts. The download for the Xcode command line tools can take some time.

2. Run the following commands:

   ```bash
   brew update
   brew install openssl
   brew install python@2
   ```

   The AWS IoT Device SDK for Python requires OpenSSL version 1.0.1 (or later) compiled with the Python executable. The `brew install python` command installs a python2 executable that meets this requirement. The python2 executable is
installed in the /usr/local/bin directory, which should be part of the PATH environment variable. To confirm, run the following command:

```
python2 --version
```

If python2 version information is provided, skip to the next step. Otherwise, permanently add the /usr/local/bin path to your PATH environment variable by appending the following line to your shell profile:

```
export PATH="/usr/local/bin:$PATH"
```

For example, if you're using .bash_profile or do not yet have a shell profile, run the following command from a Terminal window:

```
echo 'export PATH="/usr/local/bin:$PATH"' >> ~/.bash_profile
```

Next, source your shell profile and confirm that python2 --version provides version information. For example, if you're using .bash_profile, run the following commands:

```
source ~/.bash_profile
python2 --version
```

python2 version information should be returned.

3. Append the following line to your shell profile:

```
alias python="python2"
```

For example, if you're using .bash_profile or do not yet have a shell profile, run the following command:

```
echo 'alias python="python2"' >> ~/.bash_profile
```

4. Next, source your shell profile. For example, if you're using .bash_profile, run the following command:

```
source ~/.bash_profile
```

Invoking the python command runs the Python executable that contains the required OpenSSL version (python2).

5. Run the following commands:

```
python
import ssl
print ssl.OPENSSL_VERSION
```

The OpenSSL version should be 1.0.1 or later.

6. To exit the Python shell, run the following command:

```
exit()
```
Install the AWS IoT Device SDK for Python

**UNIX-like system**

1. From a terminal window, run the following command:

   ```
   python --version
   ```

   If no version information is returned or if the version number is less than 2.7 for Python 2 or less than 3.3 for Python 3, follow the instructions in Downloading Python to install Python 2.7+ or Python 3.3+. For more information, see Using Python on Unix platforms.

2. In the terminal, run the following commands to determine the OpenSSL version:

   ```
   python
   >>>import ssl
   >>>print ssl.OPENSSL_VERSION
   ```

   Make a note of the OpenSSL version value.

   To close the Python shell, run the following command:

   ```
   exit()
   ```

   If the OpenSSL version is 1.0.1 or later, skip to the next step. Otherwise, run the command(s) to update OpenSSL for your distribution (for example, `sudo yum update openssl`, `sudo apt-get update`, and so on).

   Confirm that the OpenSSL version is 1.0.1 or later by running the following commands:

   ```
   python
   >>>import ssl
   >>>print ssl.OPENSSL_VERSION
   >>>exit()
   ```

3. Run the following commands to install the AWS IoT Device SDK for Python:

   ```
   cd ~
   git clone https://github.com/aws/aws-iot-device-sdk-python.git
   cd aws-iot-device-sdk-python
   python setup.py install
   ```

2. After the AWS IoT Device SDK for Python is installed, navigate to the `samples` folder, open the `greengrass` folder, and then copy the `basicDiscovery.py` file to the folder that contains the `HelloWorld_Publisher` and `HelloWorld_Subscriber` device certificates files, as shown in the following example. (The hash component in your file names are different.)
Test Communications

1. Make sure that your computer and the AWS IoT Greengrass core device are connected to the internet using the same network.

   a. On the AWS IoT Greengrass core device, run the following command to find its IP address.

      ```
      hostname -I
      ```

   b. On your computer, run the following command using the IP address of the core. You can use Ctrl + C to stop the ping command.

      ```
      ping IP-address
      ```

      Output similar to the following indicates successful communication between the computer and the AWS IoT Greengrass core device (0% packet loss):

      ```
      PING 176.32.103.205 (176.32.103.205) 56(84) bytes of data.
      64 bytes from 176.32.103.205: icmp_seq=1 ttl=230 time=77.2 ms
      64 bytes from 176.32.103.205: icmp_seq=2 ttl=230 time=77.1 ms
      64 bytes from 176.32.103.205: icmp_seq=3 ttl=230 time=77.1 ms
      64 bytes from 176.32.103.205: icmp_seq=4 ttl=230 time=77.1 ms
      64 bytes from 176.32.103.205: icmp_seq=5 ttl=230 time=77.1 ms
      64 bytes from 176.32.103.205: icmp_seq=6 ttl=230 time=77.1 ms
      ```

      Note

      If you’re unable to ping an EC2 instance that’s running AWS IoT Greengrass, make sure that the inbound security group rules for the instance allow ICMP traffic for Echo Request messages. For more information, see Adding Rules to a Security Group in the Amazon EC2 User Guide for Linux Instances.

      On Windows host computers, in the Windows Firewall with Advanced Security app, you might also need to enable an inbound rule that allows inbound echo requests (for example, File and Printer Sharing (Echo Request - ICMPv4-In)), or create one.

2. Get your AWS IoT endpoint.
a. In the AWS IoT console, in the navigation pane, choose Settings.
b. Under Settings, make a note of the value of Endpoint. You use this value to replace the AWS_IOT_ENDPOINT placeholder in the commands in the following steps.

Custom endpoint

This is your custom endpoint that allows you to connect to AWS IoT. Each of your Thing
This is also an important property to insert when using an MQTT client or the AWS IoT

Your endpoint is provisioned and ready to use. You can now start to publish and sub

Endpoint

abcdefghij1289-ats.iot.us-west-2.amazonaws.com

Note
Make sure that your endpoints correspond to your certificate type (p. 45).

3. On your computer (not the AWS IoT Greengrass core device), open two command-line (terminal or command prompt) windows. One window represents the HelloWorld_Publisher device and the other represents the HelloWorld_Subscriber device.

Upon execution, basicDiscovery.py attempts to collect information on the location of the AWS IoT Greengrass core at its endpoints. This information is stored after the device has discovered and successfully connected to the core. This allows future messaging and operations to be executed locally (without the need for an internet connection).

Note
You can run the following command from the folder that contains the basicDiscovery.py file for detailed script usage information:

```
python basicDiscovery.py --help
```

4. From the HelloWorld_Publisher device window, run the following commands.

- Replace path-to-certs-folder with the path to the folder that contains the certificates, keys, and basicDiscovery.py.
- Replace AWS_IOT_ENDPOINT with your endpoint.
- Replace the two publisher instances with the hash in the file name for your HelloWorld_Publisher device.

```
cd path-to-certs-folder
python basicDiscovery.py --endpoint AWS_IOT_ENDPOINT --rootCA root-ca-cert.pem --cert publisher.cert.pem --key publisher.private.key --thingName HelloWorld_Publisher --topic 'hello/world/pubsub' --mode publish --message 'Hello, World! Sent from HelloWorld_Publisher'
```

You should see output similar to the following, which includes entries such as Published topic 'hello/world/pubsub': {"message": "Hello, World! Sent from HelloWorld_Publisher", "sequence": 1}. 
5. From the HelloWorld_Subscriber device window, run the following commands.

   - Replace `path-to-certs-folder` with the path to the folder that contains the certificates, keys, and `basicDiscovery.py`.
   - Replace `AWS_IOT_ENDPOINT` with your endpoint.
   - Replace the two `subscriber` instances with the hash in the file name for your HelloWorld_Subscriber device.

```
cd path-to-certs-folder
python basicDiscovery.py --endpoint AWS_IOT_ENDPOINT --rootCA root-ca-cert.pem --cert subscriber.cert.pem --key subscriber.private.key --thingName HelloWorld_Subscriber --topic 'hello/world/pubsub' --mode subscribe
```

You should see the following output, which includes entries such as Received message on topic hello/world/pubsub: {"message": "Hello, World! Sent from HelloWorld_Publisher", "sequence": 1}.

Close the HelloWorld_Publisher window to stop messages from accruing in the HelloWorld_Subscriber window.

Testing on a corporate network might interfere with connecting to the core. As a workaround, you can manually enter the endpoint. This ensures that the `basicDiscovery.py` script connects to the correct IP address of the AWS IoT Greengrass core device.

**To manually enter the endpoint**

1. Choose Greengrass, choose Groups, and then choose your group.
2. Choose Settings.
3. For **Local connection detection**, choose **Manually manage connection information**, and then choose **View Cores for specific endpoint information**.

4. Choose your core, and then choose **Connectivity**.

5. Choose **Edit** and make sure that you have only one endpoint value. This value must be the IP address endpoint for port 8883 of your AWS IoT Greengrass core device (for example, **192.168.1.4**).

6. Choose **Update**.

---

**Module 5: Interacting with Device Shadows**

This advanced module shows you how AWS IoT Greengrass devices can interact with AWS IoT device shadows in an AWS IoT Greengrass group. A *shadow* is a JSON document that is used to store current or desired state information for a thing. In this module, you discover how one AWS IoT Greengrass device (GG_Switch) can modify the state of another AWS IoT Greengrass device (GG_TrafficLight) and how these states can be synced to the AWS IoT Greengrass cloud:

Before you begin, make sure that you have completed Module 1 (p. 65), Module 2 (p. 76), Module 3 (Part 1) (p. 84), and Module 3 (Part 2) (p. 97). You should also understand how to connect devices to an AWS IoT Greengrass core (Module 4 (p. 109)). You do not need other components or devices.

This module should take about 30 minutes to complete.

**Topics**

- Configure Devices and Subscriptions (p. 123)
Configure Devices and Subscriptions

Shadows can be synced to AWS IoT when the AWS IoT Greengrass core is connected to the internet. In this module, you first use local shadows without syncing to the cloud. Then, you enable cloud syncing.

Each device has its own shadow. For more information, see Device Shadow Service for AWS IoT in the AWS IoT Developer Guide.

1. From the Devices page, add two new devices in your AWS IoT Greengrass group. For detailed steps of this process, see the section called “Create AWS IoT Devices in an AWS IoT Greengrass Group” (p. 110).
   - Name the devices **GG_Switch** and **GG_TrafficLight**.
   - Generate and download the 1-Click default security resources for both devices.
   - Make a note of the hash component in the file names of the security resources for the devices. You use these values later.

2. Decompress the downloaded certificates and keys for both devices into a single folder on your computer. For example, run the following command for each .tar.gz file.

   ```
   tar -xzf hash-setup.tar.gz
   ```

   **Note**
   On Windows, you can decompress .tar.gz files using a tool such as 7-Zip or WinZip.

3. Copy the `root-ca-cert.pem` file that you downloaded in the previous module (p. 113) to this folder.

4. Make sure that the devices are set to use local shadows. If not, choose the ellipsis (…), and then choose **Make local only**.
5. The function code used in this module requires that you manually configure the core's endpoint.
   a. On the group configuration page, choose **Settings**.
   b. For **Local connection detection**, choose **Manually manage connection information**, and then choose **View Cores for specific endpoint information**.
   c. Choose your core, and then choose **Connectivity**.
   d. Choose **Edit** and make sure that you have only one endpoint value. This value must be the IP address endpoint for port 8883 of your AWS IoT Greengrass core device (for example, 192.168.1.4).
   e. Choose **Update**.

6. Add the subscriptions in the following table to your group. For example, to create the first subscription:
   a. On the group configuration page, choose **Subscriptions**, and then choose **Add subscription**.
   b. Under **Select a source**, choose **Devices**, and then choose **GG_Switch**.
   c. Under **Select a target**, choose **Services**, and then choose **Local Shadow Service**.
   d. Choose **Next**.
   e. For **Topic filter**, enter $aws/things/GG_TrafficLight/shadow/update
   f. Choose **Next**, and then choose **Finish**.

The topics must be entered exactly as shown in the table. Although it's possible to use wildcards to consolidate some of the subscriptions, we don't recommend this practice. For more information, see **Shadow MQTT Topics** in the **AWS IoT Developer Guide**.

<table>
<thead>
<tr>
<th>Source</th>
<th>Target</th>
<th>Topic</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>GG_Switch</td>
<td>Local Shadow Service</td>
<td>$aws/things/GG_TrafficLight/shadow/update</td>
<td>The GG_Switch sends an update request to update topic.</td>
</tr>
<tr>
<td>Local Shadow Service</td>
<td>GG_Switch</td>
<td>$aws/things/GG_TrafficLight/shadow/update/accepted</td>
<td>The GG_Switch needs to know whether the update request was accepted.</td>
</tr>
<tr>
<td>Local Shadow Service</td>
<td>GG_Switch</td>
<td>$aws/things/GG_TrafficLight/shadow/update/rejected</td>
<td>The GG_Switch needs to know whether the update request was rejected.</td>
</tr>
</tbody>
</table>
Configure Devices and Subscriptions

<table>
<thead>
<tr>
<th>Source</th>
<th>Target</th>
<th>Topic</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Shadow Service</td>
<td>GG_TrafficLight</td>
<td>$aws/things/GG_TrafficLight/shadow/update/delta</td>
<td>The Local Shadow Service sends a received update to GG_TrafficLight through the delta topic.</td>
</tr>
<tr>
<td>Local Shadow Service</td>
<td>GG_TrafficLight</td>
<td>$aws/things/GG_TrafficLight/shadow/update/accepted</td>
<td>The GG_TrafficLight needs to know whether its state update was accepted.</td>
</tr>
<tr>
<td>Local Shadow Service</td>
<td>GG_TrafficLight</td>
<td>$aws/things/GG_TrafficLight/shadow/update/rejected</td>
<td>The GG_TrafficLight needs to know whether its state update was rejected.</td>
</tr>
</tbody>
</table>

The new subscriptions are displayed on the **Subscriptions** page. To see the full topic path of a subscription, hover your mouse over the **Topic** column.

<table>
<thead>
<tr>
<th>Source</th>
<th>Target</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Shadow Service</td>
<td>GG_TrafficLight</td>
<td>$aws/things/GG_TrafficLight...</td>
</tr>
<tr>
<td>Local Shadow Service</td>
<td>GG_TrafficLight</td>
<td>$aws/things/GG_TrafficLight...</td>
</tr>
<tr>
<td>Local Shadow Service</td>
<td>GG_Switch</td>
<td>$aws/things/GG_Switch...</td>
</tr>
<tr>
<td>GG_TrafficLight</td>
<td>Local Shadow Service</td>
<td>$aws/things/GG_TrafficLight...</td>
</tr>
<tr>
<td>Local Shadow Service</td>
<td>GG_Switch</td>
<td>$aws/things/GG_TrafficLight...</td>
</tr>
<tr>
<td>GG_Switch</td>
<td>Local Shadow Service</td>
<td>$aws/things/GG_TrafficLight...</td>
</tr>
<tr>
<td>Local Shadow Service</td>
<td>GG_TrafficLight</td>
<td>$aws/things/GG_TrafficLight...</td>
</tr>
</tbody>
</table>

**Note**
For information about the $ character, see **Reserved Topics**.

7. Make sure that the AWS IoT Greengrass daemon is running, as described in **Deploy Cloud Configurations to a Core Device** (p. 93).
8. On the group configuration page, from **Actions**, choose **Deploy**.
This deploys the group configuration to your AWS IoT Greengrass core device. For troubleshooting help, see Troubleshooting (p. 516).

Download Required Files

1. If you haven’t already done so, install the AWS IoT Device SDK for Python. For instructions, see step 1 in the section called “Install the AWS IoT Device SDK for Python” (p. 114).

   This SDK is used by AWS IoT devices to communicate with AWS IoT and with AWS IoT Greengrass core devices.

2. From the AWS IoT Greengrass samples repository on GitHub, download the lightController.py and trafficLight.py files to your computer. Save them in the folder that contains the GG_Switch and GG_TrafficLight device certificates and keys.

   The lightController.py script corresponds to the GG_Switch device, and the trafficLight.py script corresponds to the GG_TrafficLight device.

Test Communications (Device Syncs Disabled)

1. Make sure that your computer and the AWS IoT Greengrass core device are connected to the internet using the same network.

   a. On the AWS IoT Greengrass core device, run the following command to find its IP address.

      ```bash
      hostname -I
      ```

   b. On your computer, run the following command using the IP address of the core. You can use Ctrl + C to stop the ping command.

      ```bash
      ping IP-address
      ```
Output similar to the following indicates successful communication between the computer and the AWS IoT Greengrass core device (0% packet loss):

```
$ping 176.32.103.205
PING 176.32.103.205 (176.32.103.205) 56(84) bytes of data.
64 bytes from 176.32.103.205: icmp_seq=1 ttl=230 time=77.2 ms
64 bytes from 176.32.103.205: icmp_seq=2 ttl=230 time=77.1 ms
64 bytes from 176.32.103.205: icmp_seq=3 ttl=230 time=77.1 ms
64 bytes from 176.32.103.205: icmp_seq=4 ttl=230 time=77.1 ms
64 bytes from 176.32.103.205: icmp_seq=5 ttl=230 time=77.1 ms
64 bytes from 176.32.103.205: icmp_seq=6 ttl=230 time=77.1 ms
^C
--- 176.32.103.205 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5549ms
rtt min/avg/max/mdev = 77.107/77.172/77.256/0.361 ms
```

**Note**

If you're unable to ping an EC2 instance that's running AWS IoT Greengrass, make sure that the inbound security group rules for the instance allow ICMP traffic for Echo Request messages. For more information, see Adding Rules to a Security Group in the Amazon EC2 User Guide for Linux Instances.

On Windows host computers, in the Windows Firewall with Advanced Security app, you might also need to enable an inbound rule that allows inbound echo requests (for example, File and Printer Sharing (Echo Request - ICMPv4-In)), or create one.

2. Get your AWS IoT endpoint.
   a. In the AWS IoT console, in the navigation pane, choose Settings.
   b. Under Settings, make a note of the value of Endpoint. You use this value to replace the AWS_IOT_ENDPOINT placeholder in the commands in the following steps.

**Custom endpoint**

This is your custom endpoint that allows you to connect to AWS IoT. Each of your Thing
This is also an important property to insert when using an MQTT client or the AWS IoT!

Your endpoint is provisioned and ready to use. You can now start to publish and sub

**Endpoint**

```
abcdefghij1289-ats.iot.us-west-2.amazonaws.com
```

**Note**

Make sure that your endpoints correspond to your certificate type (p. 45).

3. On your computer (not the AWS IoT Greengrass core device), open two command-line (terminal or command prompt) windows. One window represents the GG_Switch device and the other represents the GG_TrafficLight device.
   a. From the GG_Switch device window, run the following commands.
      * Replace path-to-certs-folder with the path to the folder that contains the certificates, keys, and Python files.
• Replace `AWS_IOT_ENDPOINT` with your endpoint.

• Replace the two `switch` instances with the hash in the file name for your GG_Switch device.

```
cd path-to-certs-folder
python lightController.py --endpoint AWS_IOT_ENDPOINT --rootCA root-ca-cert.pem --cert switch.cert.pem --key switch.private.key --thingName GG_TrafficLight --clientId GG_Switch
```

b. From the GG_TrafficLight device window, run the following commands.

• Replace `path-to-certs-folder` with the path to the folder that contains the certificates, keys, and Python files.

• Replace `AWS_IOT_ENDPOINT` with your endpoint.

• Replace the two `light` instances with the hash in the file name for your GG_TrafficLight device.

```
cd path-to-certs-folder
python trafficLight.py --endpoint AWS_IOT_ENDPOINT --rootCA root-ca-cert.pem --cert light.cert.pem --key light.private.key --thingName GG_TrafficLight --clientId GG_TrafficLight
```

Every 20 seconds, the switch updates the shadow state to G, Y, and R, and the light displays its new state, as shown next.

**GG_Switch output:**

```
{
state: {
"desired": {
"property": ["R"]
},
2018-12-20 12:13:01.469 - AWSIoTPythonSDK.core.protocol.mqtt.core - INFO - Performing sync publish...
---------- Shadow Update Accepted ------------
Update request with token: 3db2de7c-93bd-4e6a-8562-0f6888240f4 accepted!
property: R

```

**GG_TrafficLight output:**

```
********* Received Shadow Delta *********
{
state: [u'property': [u'R']]
}, u'metadata': [{u'property': [u'timestamp': 1544937831]}, u'version': 33, u'clientToken': u'3db2de7c-93bd-4e6a-8562-0f6888240f4']
property: R
version: 33

*************

Light changed to: R

3db2010f-c1c2-4aeb-a841-8440500eefc accepted!
property: R
```

When executed for the first time, each device script runs the AWS IoT Greengrass discovery service to connect to the AWS IoT Greengrass core (through the internet). After a device has discovered and successfully connected to the AWS IoT Greengrass core, future operations can be executed locally.

**Note**

The `lightController.py` and `trafficLight.py` scripts store connection information in the `groupCA` folder, which is created in the same folder as the scripts. If you receive connection errors, make sure that the IP address in the `ggc-host` file matches the single IP address endpoint that you configured for your core in [this step](p. 124).
4. In the AWS IoT console, choose your AWS IoT Greengrass group, choose Devices, and then choose GG_TrafficLight.

5. Choose Shadow. After the GG_Switch changes states, there should not be any updates to this shadow topic in Shadow State. That's because the GG_TrafficLight is set to LOCAL SHADOW ONLY as opposed to SHADOW SYNCING TO CLOUD.

6. Press Ctrl + C in the GG_Switch (lightController.py) device window. You should see that the GG_TrafficLight (trafficLight.py) window stops receiving state change messages.

Keep these windows open so you can run the commands in the next section.

Test Communications (Device Syncs Enabled)

For this test, you configure the GG_TrafficLight device shadow to sync to AWS IoT. You run the same commands as in the previous test, but this time the shadow state in the cloud is updated when GG_Switch sends an update request.

1. In the AWS IoT console, choose your AWS IoT Greengrass group, and then choose Devices.
2. For the GG_TrafficLight device, choose the ellipsis (...), and then choose Sync to the Cloud.

You should receive a notification that the device shadow was updated.

3. On the group configuration page, from Actions, choose Deploy.
This deploys the group configuration to your AWS IoT Greengrass core device. For troubleshooting help, see Troubleshooting (p. 516).

4. In your two command-line windows, run the commands from the previous test for the GG_Switch (p. 127) and GG_TrafficLight (p. 128) devices.

5. Now, check the shadow state in the AWS IoT console. Choose your AWS IoT Greengrass group, choose Devices, choose GG_TrafficLight, and then choose Shadow.

Because you enabled sync of the GG_TrafficLight shadow to AWS IoT, the shadow state in the cloud should be updated whenever GG_Switch sends an update. This functionality can be used to expose the state of a Greengrass device to AWS IoT.

**Shadow Document**

**Last update:** Jan 9, 2018 3:39:53 PM -0800

**Shadow state:**
```json
1 { 
2   "desired": { 
3     "property": "G"
4   }, 
5   "reported": { 
6     "property": "G"
7   }
8 }
```

**Shadow Document**

**Last update:** Jan 9, 2018 3:51:14 PM -0800

**Shadow state:**
```json
1 { 
2   "desired": { 
3     "property": "Y"
4   }, 
5   "reported": { 
6     "property": "Y"
7   }
8 }
```

**Note**

If necessary, you can troubleshoot issues by viewing the AWS IoT Greengrass core logs, particularly runtime.log:

```
cd /greengrass/ggc/var/log
```
sudo cat system/runtime.log | more

You can also view GGShadowSyncManager.log and GGShadowService.log. For more information, see Troubleshooting (p. 516).

Keep the devices and subscriptions set up. You use them in the next module. You also run the same commands.

**Module 6: Accessing Other AWS Services**

This advanced module shows you how AWS IoT Greengrass cores can interact with other AWS services in the cloud. It builds on the traffic light example from Module 5 (p. 122) and adds a Lambda function that processes shadow states and uploads a summary to an Amazon DynamoDB table.

Before you begin, make sure that you have completed Module 1 (p. 65) through Module 5 (p. 122). You do not need other components or devices.

This module should take approximately 30 minutes to complete.
**Note**
This module creates and updates a table in DynamoDB. Although most of the operations are small and fall within the AWS Free Tier, performing some of the steps in this module might result in charges to your account. For information about pricing, see DynamoDB pricing documentation.

**Topics**
- Configure the Group Role (p. 132)
- Create and Configure the Lambda Function (p. 134)
- Configure Subscriptions (p. 138)
- Test Communications (p. 140)

**Configure the Group Role**

The group role is an IAM role that you create and attach to your group. This role contains the permissions that your deployed Lambda functions (and connectors (p. 283)) use to access AWS services. For more information about IAM roles, see the IAM User Guide.

In this step, you create a permissions policy that allows describe, create, and update actions on an Amazon DynamoDB table. Then, you attach the policy to a new role and associate the role with your Greengrass group.

First, create a customer-managed policy that grants permissions required by the Lambda function in this module.

1. In the IAM console, in the navigation pane, choose **Policies**, and then choose **Create policy**.
2. On the **JSON** tab, replace the placeholder content with the following policy. The Lambda function in this module uses these permissions to create and update a DynamoDB table named CarStats.

   ```json
   {
     "Version": "2012-10-17",
     "Statement": [
       {
         "Sid": "PermissionsForModule6",
         "Effect": "Allow",
         "Action": [
           "dynamodb:DescribeTable",
           "dynamodb:CreateTable",
           "dynamodb:PutItem"
         ],
         "Resource": "arn:aws:dynamodb:*:*:table/CarStats"
       }
     ]
   }
   ```
3. Choose **Review policy**.
4. For **Name**, enter `greengrass_CarStats_Table`, and then choose **Create policy**.

Next, create a role that uses the new policy.

5. In the navigation pane, choose **Roles**, and then choose **Create role**.
6. Under **Select type of trusted entity**, choose AWS service.
7. Under **Choose the service that will use this role**, choose Greengrass, and then choose **Next: Permissions**.
8. Under **Attach permissions policies**, select the new `greengrass_CarStats_Table` policy.
9. Choose **Next: Tags**, and then choose **Next: Review**. Tags aren't used in this tutorial.

10. For **Role name**, enter **Greengrass_Group_Role**.

11. For **Role description**, enter **Greengrass group role for connectors and user-defined Lambda functions**.

   **Review**
   
   Provide the required information below and review this role before you create it.
   
   **Role name**
   
   Use alphanumeric and `-._@` characters. Maximum 64 characters.
   
   **Role description**
   
   Greengrass group role for connectors and user-defined Lambda functions.
   
   Maximum 1000 characters. Use alphanumeric and `-._@` characters.

   **Trusted entities**
   
   AWS service: greengrass.amazonaws.com

   **Policies**
   
   greengrass_CarStats_Table

   **Permissions boundary**
   
   Permissions boundary is not set

   **No tags were added.**

12. Choose **Create role**.

   Now, attach the role to your Greengrass group.

13. In the AWS IoT console, under **Greengrass**, choose **Groups**, and then choose your AWS IoT Greengrass group.

14. Choose **Settings**, and then choose **Add Role**.
15. Choose **Greengrass_Group_Role** from your list of roles, and then choose **Save**.

**Create and Configure the Lambda Function**

In this step, you create a Lambda function that tracks the number of cars that pass the traffic light. Every time that the GG_TrafficLight shadow state changes to G, the Lambda function simulates the passing of a random number of cars (from 1 to 20). On every third G light change, the Lambda function sends basic statistics, such as min and max, to a DynamoDB table.

1. On your computer, create a folder named **car_aggregator**.
2. From the [AWS IoT Greengrass samples](https://github.com/aws-iot-greengrass) repository on GitHub, download the **carAggregator.py** function to the **car_aggregator** folder. This is your Lambda function code.
3. Run the following command in a command-line window to install the **Boto 3 - The AWS SDK for Python** package and its dependencies in the **car_aggregator** folder. Greengrass Lambda functions use the AWS SDK to access other AWS services. (For Windows, use an elevated command prompt.)

   ```shell
   pip install boto3 -t path-to-car_aggregator-folder
   ```

   This results in a directory listing similar to the following:
4. Compress the contents of the car_aggregator folder into a .zip file named car_aggregator.zip. (Compress the folder's contents, not the folder.) This is your Lambda function deployment package.

5. In the Lambda console, create a function named **GG_Car_Aggregator**, and set the remaining fields as follows:
   - For **Runtime**, choose **Python 2.7**.
   - For **Permissions**, keep the default setting. This creates an execution role that grants basic Lambda permissions. This role isn't used by AWS IoT Greengrass.

Choose **Create function**.
6. Upload your Lambda function deployment package:
   a. On the **Configuration** tab, under **Function code**, set the following fields:
      - For **Code entry type**, choose **Upload a .zip file**.
      - For **Runtime**, choose **Python 2.7**.
      - For **Handler**, enter **carAggregator.function_handler**
   b. Choose **Upload**, and then choose car_aggregator.zip.
   c. Choose **Save**.

7. Publish the Lambda function, and then create an alias named **GG_CarAggregator**. For step-by-step instructions, see the steps to publish the Lambda function (p. 87) and create an alias (p. 87) in Module 3 (Part 1).
8. In the AWS IoT console, add the Lambda function that you just created to your AWS IoT Greengrass group:
   a. On the group configuration page, choose Lambda, and then choose Add Lambda.
   b. Choose Use existing Lambda.
   c. Choose GG_Car_Aggregator, and then choose Next.
   d. Choose Alias: GG_CarAggregator, and then choose Finish.
9. Edit the Lambda function configuration:
   a. Choose the ellipsis (…) associated with the Lambda function, and then choose Edit Configuration.

   b. Under Memory limit, enter 64 MB.
   c. Under Lambda lifecycle, choose Make this function long-lived and keep it running indefinitely, and then choose Update.

Configure Subscriptions

In this step, you create a subscription that enables the GG_TrafficLight shadow to send updated state information to the GG_Car_Aggregator Lambda function. This subscription is added to the subscriptions that you created in Module 5 (p. 122), which are all required for this module.

1. On the group configuration page, choose Subscriptions, and then choose Add Subscription.
2. On the Select your source and target page, set the following values:
   - For Select a source, choose Services, and then choose Local Shadow Service.
   - For Select a target, choose Lambdas, and then choose GG_Car_Aggregator.

Choose Next.
3. On the Filter your data with a topic page, for Topic filter, enter the following topic:

   $aws/things/GG_TrafficLight/shadow/update/documents

4. Choose Next, and then choose Finish.

   This module requires the new subscription and the subscriptions (p. 124) that you created in Module 5.

5. Make sure that the AWS IoT Greengrass daemon is running, as described in Deploy Cloud Configurations to a Core Device (p. 93).

6. On the group configuration page, from Actions, choose Deploy.

   This deploys the group configuration to your AWS IoT Greengrass core device. For troubleshooting help, see Troubleshooting (p. 516).
Test Communications

1. On your computer, open two command-line windows. Just as in Module 5 (p. 122), one window is for the GG_Switch device and the other is for the GG_TrafficLight device. You use them to run the same commands that you ran in Module 5.

Run the following commands for the GG_Switch device:

```bash
cd path-to-certs-folder
python lightController.py --endpoint AWS_IOT_ENDPOINT --rootCA root-ca-cert.pem --cert switch.cert.pem --key switch.private.key --thingName GG_TrafficLight --clientId GG_Switch
```

Run the following commands for the GG_TrafficLight device:

```bash
cd path-to-certs-folder
python trafficLight.py --endpoint AWS_IOT_ENDPOINT --rootCA root-ca-cert.pem --cert light.cert.pem --key light.private.key --thingName GG_TrafficLight --clientId GG_TrafficLight
```

Every 20 seconds, the switch updates the shadow state to G, Y, and R, and the light displays its new state.

2. The function handler of the Lambda function is triggered on every third green light (every three minutes), and a new DynamoDB record is created. After `lightController.py` and `trafficLight.py` have run for three minutes, go to the AWS Management Console, and open the DynamoDB console.

3. Choose **US East (N. Virginia)** in the AWS Region menu. This is the Region where the **GG_Car_Aggregator** function creates the table.

4. In the navigation pane, choose **Tables**, and then choose the **CarStats** table.

On the **Items** tab, you should see entries with basic statistics on cars passed (one entry for every three minutes). You might need to choose the refresh button to view updates to the table.
5. If the test is not successful, you can look for troubleshooting information in the Greengrass logs.
   a. Switch to the root user and navigate to the log directory. Access to AWS IoT Greengrass logs requires root permissions.

   ```
   sudo su
   cd /greengrass/ggc/var/log
   ```

   b. Check `runtime.log` for errors.

   ```
   cat system/runtime.log | grep 'ERROR'
   ```

c. Check the log generated by the Lambda function.

   ```
   cat user/region/account-id/GG_Car_Aggregator.log
   ```

   The `lightController.py` and `trafficLight.py` scripts store connection information in the `groupCA` folder, which is created in the same folder as the scripts. If you receive connection errors, make sure that the IP address in the `ggc-host` file matches the single IP address endpoint that you configured for your core in this step (p. 124).

   For more information, see Troubleshooting (p. 516).
This is the end of the basic tutorial. You should now understand the AWS IoT Greengrass programming model and its fundamental concepts, including AWS IoT Greengrass cores, groups, subscriptions, devices, and the deployment process for Lambda functions running at the edge.

You can delete the DynamoDB table and the Greengrass Lambda functions and subscriptions. To stop communications between the AWS IoT Greengrass core device and the AWS IoT cloud, open a terminal on the core device and run one of the following commands:

- To shut down the AWS IoT Greengrass core device:
  ```
  sudo halt
  ```

- To stop the AWS IoT Greengrass daemon:
  ```
  cd /greengrass/ggc/core/
  sudo ./greengrassd stop
  ```

### Module 7: Simulating Hardware Security Integration

This feature is available for AWS IoT Greengrass Core v1.7 and later.

This advanced module shows you how to configure a simulated hardware security module (HSM) for use with a Greengrass core. The configuration uses SoftHSM, which is a pure software implementation that uses the PKCS#11 (p. 147) application programming interface (API). The purpose of this module is to allow you to set up an environment where you can learn and do initial testing against a software-only implementation of the PKCS#11 API. It is provided only for learning and initial testing, not for production use of any kind.

You can use this configuration to experiment with using a PKCS#11-compatible service to store your private keys. For more information about the software-only implementation, see SoftHSM. For more information about integrating hardware security on an AWS IoT Greengrass core, including general requirements, see the section called “Hardware Security” (p. 448).

**Important**

This module is intended for experimentation purposes only. We strongly discourage the use of SoftHSM in a production environment because it might provide a false sense of additional security. The resulting configuration doesn’t provide any actual security benefits. The keys stored in SoftHSM are not stored more securely than any other means of secrets storage in the Greengrass environment.

The purpose of this module is to allow you to learn about the PKCS#11 specification and do initial testing of your software if you plan to use a real hardware-based HSM in the future. You must test your future hardware implementation separately and completely before any production usage because there might be differences between the PKCS#11 implementation provided in SoftHSM and a hardware-based implementation.

If you need assistance with the onboarding of a supported hardware security module (p. 449), contact your AWS Enterprise Support representative.

Before you begin, make sure that you completed Module 1 (p. 65) and Module 2 (p. 76) of the Getting Started tutorial. In this module, we assume that your core is already provisioned and communicating with AWS. This module should take about 30 minutes to complete.

### Install the SoftHSM Software

In this step, you install SoftHSM and the pkcs11 tools, which are used to manage your SoftHSM instance.
In a terminal on your AWS IoT Greengrass core device, run the following command:

```bash
sudo apt-get install softhsm2 libsofthsm2-dev pkcs11-dump
```

For more information about these packages, see Install softhsm2, Install libsofthsm2-dev, and Install pkcs11-dump.

**Note**
If you encounter issues when using this command on your system, see SoftHSM version 2 on GitHub. This site provides more installation information, including how to build from source.

## Configure SoftHSM

In this step, you configure SoftHSM.

1. Switch to the root user.

   ```bash
   sudo su
   ``)

2. Use the manual page to find the system-wide softhsm2.conf location. A common location is /etc/softhsm/softhsm2.conf, but the location might be different on some systems.

   ```bash
   man softhsm2.conf
   ``)

3. Create the directory for the softhsm2 configuration file in the system-wide location. In this example, we assume the location is /etc/softhsm/softhsm2.conf.

   ```bash
   mkdir -p /etc/softhsm
   ``)

4. Create the token directory in the /greengrass directory.

   **Note**
   If this step is skipped, softhsm2-util reports ERROR: Could not initialize the library.

   ```bash
   mkdir -p /greengrass/softhsm2/tokens
   ``)

5. Configure the token directory.

   ```bash
   echo "directories.tokendir = /greengrass/softhsm2/tokens" > /etc/softhsm/softhsm2.conf
   ``)

6. Configure a file-based backend.

   ```bash
   echo "objectstore.backend = file" >> /etc/softhsm/softhsm2.conf
   ``)

   **Note**
   These configuration settings are intended for experimentation purposes only. To see all configuration options, read the manual page for the configuration file.

   ```bash
   man softhsm2.conf
   ```
Import the Private Key

In this step, you initialize the SoftHSM token, convert the private key format, and then import the private key.

1. Initialize the SoftHSM token.
   
   ```
   softhsm2-util --init-token --slot 0 --label greengrass --so-pin 12345 --pin 1234
   ```

   **Note**
   
   If prompted, enter an SO pin of 12345 and a user pin of 1234. AWS IoT Greengrass doesn't use the SO (supervisor) pin, so you can use any value.

2. Convert the private key to a format that can be used by the SoftHSM import tool. For this tutorial, you convert the private key that you obtained from the Easy Group creation option in Module 2 (p. 76) of the Getting Started tutorial.
   
   ```
   openssl pkcs8 -topk8 -inform PEM -outform PEM -nocrypt -in hash.private.key -out hash.private.pem
   ```

3. Import the private key into SoftHSM. Run only one of the following commands, depending on your version of softhsm2-util.

   **Raspbian softhsm2-util v2.2.0 syntax**
   
   ```
   softhsm2-util --import hash.private.pem --token greengrass --label iotkey --id 0000 --pin 12340
   ```

   **Ubuntu softhsm2-util v2.0.0 syntax**
   
   ```
   softhsm2-util --import hash.private.pem --slot 0 --label iotkey --id 0000 --pin 1234
   ```

   This command identifies the slot as 0 and defines the key label as iotkey. You use these values in the next section.

After the private key is imported, you can optionally remove it from the `/greengrass/certs` directory. Make sure to keep the root CA and device certificates in the directory.

Configure the Greengrass Core to Use SoftHSM

In this step, you modify the Greengrass core configuration file to use SoftHSM.

1. Find the path to the SoftHSM provider library (`libsofthsm2.so`) on your system:
   
   a. Get the list of installed packages for the library.
      
      ```
      sudo dpkg -L libsofthsm2
      ```

      The `libsofthsm2.so` file is located in the `softhsm` directory.

   b. Copy the full path to the file (for example, `/usr/lib/x86_64-linux-gnu/softhsm/libsofthsm2.so`). You use this value later.

   2. Stop the AWS Greengrass daemon.
cd /greengrass/ggc/core/
sudo ./greengrassd stop

3. Open the Greengrass configuration file. This is the `config.json` file in the `/greengrass/config` directory.

   **Note**
   The examples in this procedure are written with the assumption that the `config.json` file uses the format that's generated from the Easy Group creation option in Module 2 (p. 76) of the Getting Started tutorial.

4. In the `crypto.principals` object, insert the following MQTT server certificate object. Add a comma where needed to create a valid JSON file.

   ```json
   "MQTTServerCertificate": {
       "privateKeyPath": "path-to-private-key"
   }
   ```

5. In the `crypto` object, insert the following PKCS11 object. Add a comma where needed to create a valid JSON file.

   ```json
   "PKCS11": {
       "P11Provider": "/path-to-pkcs11-provider-so",
       "slotLabel": "crypto-token-name",
       "slotUserPin": "crypto-token-user-pin"
   }
   ```

   Your file should look similar to the following:

   ```json
   {
   "coreThing": {
       "caPath": "root.ca.pem",
       "certPath": "hash.cert.pem",
       "keyPath": "hash.private.key",
       "thingArn": "arn:partition:iot:region:account-id:thing/core-thing-name",
       "iotHost": "host-prefix.iot.region.amazonaws.com",
       "ggHost": "greengrass.iot.region.amazonaws.com",
       "keepAlive": 600
   },
   "runtime": {
       "cgroup": {
           "useSystemd": "yes"
       }
   },
   "managedRespawn": false,
   "crypto": {
       "PKCS11": {
           "P11Provider": "/path-to-pkcs11-provider-so",
           "slotLabel": "crypto-token-name",
           "slotUserPin": "crypto-token-user-pin"
       },
       "principals": {
           "MQTTServerCertificate": {
               "privateKeyPath": "path-to-private-key"
           },
           "IoTCertificate": {
               "privateKeyPath": "file:///greengrass/certs/hash.private.key",
               "certificatePath": "file:///greengrass/certs/hash.cert.pem"
           },
           "SecretsManager": {
               "privateKeyPath": "file:///greengrass/certs/hash.private.key"
           }
       }
   }
   ```
Note
To use over-the-air (OTA) updates with hardware security, the PKCS11 object must also contain the OpenSSLEngine property. For more information, see the section called “Configure OTA Updates” (p. 454).

6. Edit the crypto object:
   a. Configure the PKCS11 object.
      - For P11Provider, enter the full path to libsofthsm2.so.
      - For slotLabel, enter greengrass.
      - For slotUserPin, enter 1234.
   b. Configure the private key paths in the principals object. Do not edit the certificatePath property.
      - For the privateKeyPath properties, enter the following RFC 7512 PKCS#11 path (which specifies the key's label). Do this for the IoTCertificate, SecretsManager, and MQTTServerCertificate principals.

```
pkcs11:object=iotkey;type=private
```

   c. Check the crypto object. It should look similar to the following:

```
"crypto": {
  "PKCS11": {
    "P11Provider": "/usr/lib/x86_64-linux-gnu/softhsm/libsofthsm2.so",
    "slotLabel": "greengrass",
    "slotUserPin": "1234"
  },
  "principals": {
    "MQTTServerCertificate": {
      "privateKeyPath": "pkcs11:object=iotkey;type=private"
    },
    "SecretsManager": {
      "privateKeyPath": "pkcs11:object=iotkey;type=private"
    },
    "IoTCertificate": {
      "certificatePath": "file://certs/core.crt",
      "privateKeyPath": "pkcs11:object=iotkey;type=private"
    }
  },
  "caPath": "file://certs/root.ca.pem"
}
```

7. Remove the caPath, certPath, and keyPath values from the coreThing object. It should look similar to the following:

```
"coreThing": {
  "thingArn": "arn:partition:iot:region:account-id:thing/core-thing-name",
  "iotHost": "host-prefix-ats.iot.region.amazonaws.com",
  "ggHost": "greengrass-ats.iot.region.amazonaws.com",
  "keepAlive": 600
}
```
Note
For this tutorial, you specify the same private key for all principals. For more information about choosing the private key for the local MQTT server, see Performance (p. 453). For more information about the local secrets manager, see Deploy Secrets to the Core (p. 263).

Test the Configuration

• Start the AWS Greengrass daemon.

```bash
cd /greengrass/ggc/core/
sudo ./greengrassd start
```

If the daemon starts successfully, then your core is configured correctly.

You are now ready to learn about the PKCS#11 specification and do initial testing with the PKCS#11 API that's provided by the SoftHSM implementation.

Important
Again, it's extremely important to be aware that this module is intended for learning and testing only. It doesn't actually increase the security posture of your Greengrass environment.

Instead, the purpose of the module is to enable you to start learning and testing in preparation for using a true hardware-based HSM in the future. At that time, you must separately and completely test your software against the hardware-based HSM prior to any production usage, because there might be differences between the PKCS#11 implementation provided in SoftHSM and a hardware-based implementation.

See Also


• RFC 7512
OTA Updates of AWS IoT Greengrass Core Software

This feature is available for AWS IoT Greengrass Core v1.3 and later.

The AWS IoT Greengrass Core software is packaged with an agent that can update the core's software or the agent itself to the latest version. These updates are sent over the air (OTA). OTA updates are the recommended way to update AWS IoT Greengrass software on your AWS IoT Greengrass core devices. You can use the AWS IoT console or the CreateSoftwareUpdateJob API to start an update. By using an OTA update, you can:

- Fix security vulnerabilities.
- Address software stability issues.
- Deploy new or improved features.

You do not have to perform manual steps or have the device that is running the Core software physically present. In the event of a failed update, the OTA update agent performs a rollback.

To support OTA updates of AWS IoT Greengrass software, your Greengrass core device must:

- Have available local storage three times the amount of the core's runtime usage requirement. For more information, see AWS IoT Greengrass Core Limits in the Amazon Web Services General Reference.
- Not have trusted boot enabled in the partition that contains the Greengrass Core platform software. (The AWS IoT Greengrass core can be installed and run on a partition with trusted boot enabled, but cannot perform an OTA update.)
- Have read/write permissions on the partition that contains the Greengrass Core platform software.
- Not be configured to use a network proxy. In AWS IoT Greengrass Core v1.9.3 or later, the OTA update agent supports updates over port 443 when MQTT traffic is configured to use port 443 instead of the default port 8883. However, the OTA update agent does not support updates through a network proxy. For more information, see the section called “Connect on Port 443 or Through a Network Proxy” (p. 46).
- Have a connection to the AWS Cloud.
- Have a correctly configured AWS IoT Greengrass core and appropriate certificates.

Before you launch an OTA update of Greengrass Core software, be aware of the impact on the devices in your Greengrass group, both on the core device and on client devices connected locally to that core:

- The core shuts down during the update.
- Any Lambda functions running on the core are shut down. If those functions write to local resources, they might leave those resources in an incorrect state unless shut down properly.
- During the core's downtime, all its connections with the AWS Cloud are lost. Messages routed through the core by client devices are lost.
- Credential caches are lost.
- Queues that hold pending work for Lambda functions are lost.
- Long-lived Lambda functions lose their dynamic state information and all pending work is dropped.

The following state information is preserved during an OTA update:
• Local shadows
• Greengrass logs
• OTA update agent logs

Greengrass OTA Update Agent

The Greengrass OTA update agent is the software component on the device that handles update jobs created and deployed in the cloud. The Greengrass OTA update agent is distributed in the same software package as the AWS IoT Greengrass Core software. The agent is located in ./greengrass/ota/ota_agent/ggc-ota. It creates its logs in /var/log/greengrass/ota/ggc_ota.txt.

You can start the Greengrass OTA update agent by executing the binary manually or by integrating it as part of an init script such as a systemd service file. The binary should be run as root. When it starts, the Greengrass OTA update agent listens for Greengrass update jobs from the cloud and executes them sequentially. The Greengrass OTA update agent ignores all other IoT job types.

Do not start multiple OTA update agent instances because this might cause conflicts.

If your Greengrass core or Greengrass OTA update agent is managed by an init system, see Integration with Init Systems (p. 152) for related configurations.

CreateSoftwareUpdateJob API

The CreateSoftwareUpdateJob API creates a software update for a core or for several cores. This API can be used to update the OTA update agent and the Greengrass Core software. It makes use of AWS IoT jobs, which provide other commands to manage a software update job on a Greengrass core. For more information, see Jobs.

The following example shows how to use the CLI to create a job that updates the AWS IoT Greengrass Core software on a core device:

```bash
aws greengrass create-software-update-job \
  --update-targets-architecture x86_64 \
  --update-targets-operating-system ubuntu \
  --software-to-update core \
  --s3-url-signer-role arn:aws::iam::123456789012:role/IotS3UrlPresigningRole \
  --update-agent-log-level WARN \
  --amzn-client-token myClientToken1
```

The create-software-update-job command returns a JSON response that contains the job ID, job ARN, and software version that was installed by the update:

```json
{
  "IotJobId": "Greengrass-OTA-c3bd7f36-ee80-4d42-8321-alda0e5b1303",
  "IotJobArn": "arn:aws::iot:region:123456789012:job/Greengrass-OTA-c3bd7f36-ee80-4d42-8321-alda0e5b1303",
  "PlatformSoftwareVersion": "1.9.2"
}
```

The create-software-update-job command has the following parameters:

--update-targets-architecture

The architecture of the core device. Must be one of armv7l, armv6l, x86_64, or aarch64.
--update-targets

A list of the targets to which the OTA update should be applied. The list can contain the ARNs of things that are cores, and the ARNs of thing groups whose members are cores. For more information, see IoT Thing Groups.

--update-targets-operating-system

The operating system of the core device. Must be one of ubuntu, amazon_linux, raspbian, or openwrt.

--software-to-update

Specifies whether the core's software or the OTA update agent software should be updated. Must be one of core or ota_agent.

--s3-url-signer-role

The IAM role used to presign the S3 URL that links to the AWS IoT Greengrass software update. You must provide a role that has the appropriate permissions policy attached. The following example policy allows access to AWS IoT Greengrass software updates in the specified AWS Regions:

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "AllowAccessToGreengrassOTAUpdateArtifacts",
         "Effect": "Allow",
         "Action": ["s3:GetObject"],
         "Resource": [
            "arn:aws:s3:::us-east-1-greengrass-updates/*",
            "arn:aws:s3:::us-west-2-greengrass-updates/*",
            "arn:aws:s3:::ap-northeast-1-greengrass-updates/*",
            "arn:aws:s3:::ap-southeast-2-greengrass-updates/*",
            "arn:aws:s3:::eu-central-1-greengrass-updates/*",
            "arn:aws:s3:::eu-west-1-greengrass-updates/*"
         ]
      }
   ]
}
```

**Note**
You can also use a wildcard * naming scheme for the Resource property to allow access to AWS IoT Greengrass software updates. For example, the following format allows access to software updates for all supported AWS Regions (current and future) that use the aws partition. Make sure to use the correct partitions for the AWS Regions you want to support.

```
"Resource": "arn:aws:s3:::*-greengrass-updates/*"
```

For more information, see Adding and Removing IAM Policies in the IAM User Guide.

Here is an example AssumeRole policy document with the minimum required trusted entities:

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Action": "sts:AssumeRole",
         "Principal": {
            "Service": "iot.amazonaws.com"
         }
      }
   ]
}
```
Here is an example IAM policy with the minimum permissions required to call the API:

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowCreateSoftwareUpdateJob",
            "Action": ["greengrass:CreateSoftwareUpdateJob"],
            "Effect": "Allow",
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": ["iam:PassRole"],
            "Resource": "arn:aws:s3:us-east-1:123456789012:role/IotS3UrlPresigningRole"
        },
        {
            "Effect": "Allow",
            "Action": ["iot:CreateJob"],
            "Resource": "*"
        }
    ]
}
```

**Note**

Because AWS IoT Greengrass is supported only on a subset of the architecture and operating system combinations possible with this command, `CreateSoftwareUpdateJob` rejects requests except for the following supported platforms:

- ubuntu/x86_64
- ubuntu/aarch64
- amazon_linux/x86_64
- raspbian/armv7l
- raspbian/armv6l
- openwrt/aarch64
- openwrt/armv7l
Integration with Init Systems

During an OTA update, binaries, including some that are running, are updated and restarted. This might cause conflicts if an init system is monitoring the state of either the AWS IoT Greengrass Core software or the Greengrass OTA update agent during the update. To help integrate the OTA update mechanism with your monitoring strategies, you can write shell scripts that run before and after an update. To tell the OTA update agent to run these shell scripts, you must include the "managedRespawn" : true flag in the ./greengrass/config/config.json file. For example:

```
{
    "coreThing": {
        ...
    },
    "runtime": {
        ...
    },
    "managedRespawn": true
}
```

When managedRespawn is set to true, the scripts must exist in the directory. Otherwise, the update fails. The directory tree should look like the following:

```
<greengrass_root>
|-- certs
|-- config
| |-- config.json
|-- ggc
|-- usr/scripts
| |-- ggc_pre_update.sh
| |-- ggc_post_update.sh
| |-- ota_pre_update.sh
| |-- ota_post_update.sh
|-- ota
```

OTA Self-Update with Managed Respawn

As the OTA update agent prepares to do a self-update, if managedRespawn is set to true, the OTA update agent looks in the ./greengrass/usr/scripts directory for the ota_pre_update.sh script and runs it.

After the OTA update agent completes the update, it attempts to run the ota_post_update.sh script from the ./greengrass/usr/scripts directory.

AWS IoT Greengrass Core Update with Managed Respawn

As the OTA update agent prepares to do an AWS IoT Greengrass core update, if managedRespawn is set to true, the OTA update agent looks in the ./greengrass/usr/scripts directory for the ggc_pre_update.sh script and runs it.

After the OTA update agent completes the update, it attempts to run the ggc_post_update.sh script from the ./greengrass/usr/scripts directory.
• The user-defined scripts in ./greengrass/usr/scripts should be owned by root and executable by root only.
• If managedRespawn is set to true, the scripts must exist and return a successful return code.
• If managedRespawn is set to false, the scripts do not run even if present on the device.
• A device that is the target of an update must not run two instances of the OTA update agent for the same AWS IoT thing. Doing so causes the two agents to process the same jobs, which creates conflicts.

OTA Update Agent Self-Update

Follow these steps to perform a self-update of the OTA update agent:

1. Make sure that the AWS IoT Greengrass core device is correctly provisioned with valid config.json file entries and the required certificates.
2. If the OTA update agent is managed by an init system, in the config.json file, make sure that managedRespawn property is set to true. Also, make sure the ota_pre_update.sh and ota_post_update.sh scripts are in the ./greengrass/usr/scripts directory.
3. Run ./greengrass/ota/ota_agent/ggc-ota.
4. Use the CreateSoftwareUpdateJob API to create an OTA self-update job. Make sure the --software-to-update parameter is set to ota_agent.

Greengrass Core Software Update

Follow these steps to perform an AWS IoT Greengrass Core software update:

1. Make sure that the AWS IoT Greengrass core device is correctly provisioned with valid config.json file entries and the required certificates.
2. If the AWS IoT Greengrass Core software is managed by an init system, in the config.json file, make sure that managedRespawn property is set to true. Also, make sure the ggc_pre_update.sh and ggc_post_update.sh scripts are in the ./greengrass/usr/scripts directory.
3. Run ./greengrass/ota/ota_agent/ggc-ota.
4. Use the CreateSoftwareUpdateJob API to create an update job for the Core software. Make sure the --software-to-update parameter is set to core.
Deploy AWS IoT Greengrass Groups to an AWS IoT Greengrass Core

AWS IoT Greengrass groups are used to organize entities in your edge environment. Groups also control how the entities in the group can interact with each other and with the AWS Cloud. For example, only the Lambda functions in the group are deployed for local execution and only the devices in the group can communicate using the local MQTT server.

A group must include a core (p. 24), which is an AWS IoT device that runs the AWS IoT Greengrass Core software. The core acts as an edge gateway and provides AWS IoT Core capabilities in the edge environment. Depending on your business need, you can also add the following entities to a group:

- **Devices**. Represented as things in the AWS IoT registry. These devices must run Amazon FreeRTOS or use the AWS IoT Device SDK (p. 10) or AWS IoT Greengrass Discovery API (p. 426) to get connection information for the core. Only devices that are members of the group can connect to the core.

- **Lambda functions**. User-defined serverless applications that execute code on the core. Lambda functions are authored in AWS Lambda and referenced from a Greengrass group. For more information, see Run Local Lambda Functions (p. 175).

- **Connectors**. Predefined serverless applications that execute code on the core. Connectors can provide built-in integration with local infrastructure, device protocols, AWS, and other cloud services. For more information, see Integrate with Services and Protocols Using Connectors (p. 283).

- **Subscriptions**. Defines the publishers, subscribers, and MQTT topics (or subjects) that are authorized for MQTT communication.

- **Resources**. References to local devices and volumes (p. 200), machine learning models (p. 221), and secrets (p. 263), used for access control by Greengrass Lambda functions and connectors.

- **Loggers**. Logging configurations for AWS IoT Greengrass system components and Lambda functions. For more information, see Monitoring (p. 457).

You manage your Greengrass group in the AWS Cloud and then deploy it to a core. The deployment copies the group configuration to the `group.json` file on the core device. This file is located in `greengrass-root/ggc/deployments/group`. 
Note
During a deployment, the Greengrass daemon process on the core device stops and then restarts.

Deploying Groups from the AWS IoT console

You can deploy a group and manage its deployments from the group's configuration page in the AWS IoT console.

Note
To open this page in the console, choose Greengrass and Groups, and then choose your group.

To deploy the current version of the group

• From Actions, choose Deploy.
To view the deployment history of the group

A group's deployment history includes the date and time, group version, and status of each deployment attempt.

1. From the navigation pane, choose **Deployments**.
2. To see more information about a deployment, including error messages, choose the row that contains the deployment.

To redeploy a group deployment

You might want to redeploy a deployment if the current deployment fails or revert to a different group version.

1. From the navigation pane, choose **Deployments**.
2. On the row that contains the deployment, in the **Status** column, choose the ellipsis (…), and then choose **Re-deploy**.

To reset group deployments

You might want to reset group deployments to move or delete a group or to remove deployment information. For more information, see the section called "Reset Deployments" (p. 163).

- From **Actions**, choose **Reset Deployments**.

---

Deploying Groups with the AWS IoT Greengrass API

The AWS IoT Greengrass API provides the following actions to deploy AWS IoT Greengrass groups and manage group deployments. You can call these actions from the AWS CLI, AWS IoT Greengrass API, or AWS SDK.

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CreateDeployment</strong></td>
<td>Creates a NewDeployment or Redeployment deployment.</td>
</tr>
<tr>
<td></td>
<td>You might want to redeploy a deployment if the current deployment fails.</td>
</tr>
<tr>
<td></td>
<td>Or you might want to redeploy to revert to a different group version.</td>
</tr>
<tr>
<td><strong>GetDeploymentStatus</strong></td>
<td>Returns the status of a deployment: Building,</td>
</tr>
<tr>
<td></td>
<td>InProgress, Success, or Failure.</td>
</tr>
</tbody>
</table>
Overview of the AWS IoT Greengrass Group Object Model

When programming with the AWS IoT Greengrass API, it's helpful to understand the Greengrass group object model.

Groups

In the AWS IoT Greengrass API, the top-level Group object consists of metadata and a list of GroupVersion objects. GroupVersion objects are associated with a Group by ID.

Group Versions

GroupVersion objects define group membership. Each GroupVersion references a CoreDefinitionVersion and other component versions by ARN. These references determine which entities to include in the group.
For example, to include three Lambda functions, one device, and two subscriptions in the group, the `GroupVersion` references:

- The `CoreDefinitionVersion` that contains the required core.
- The `FunctionDefinitionVersion` that contains the three functions.
- The `DeviceDefinitionVersion` that contains the device.
- The `SubscriptionDefinitionVersion` that contains the two subscriptions.

The `GroupVersion` deployed to a core device determines the entities that are available in the local environment and how they can interact.

**Group Components**

Components that you add to groups have a three-level hierarchy:

- A `Definition` that references a list of `DefinitionVersion` objects of a given type. For example, a `DeviceDefinition` references a list of `DeviceDefinitionVersion` objects.
- A `DefinitionVersion` that contains a set of entities of a given type. For example, a `DeviceDefinitionVersion` contains a list of `Device` objects.
- Individual entities that define their properties and behavior. For example, a `Device` defines the ARN of the corresponding device in the AWS IoT registry, the ARN of its device certificate, and whether its local shadow syncs automatically with the cloud.

You can add the following types of entities to a group:

- Connector
- Core
- Device
- Function
- Logger
- Resource
- Subscription
The following example `DeviceDefinition` references three `DeviceDefinitionVersion` objects that each contain multiple `Device` objects. Only one `DeviceDefinitionVersion` at a time is used in a group.

**Updating Groups**

In the AWS IoT Greengrass API, you use versions to update a group’s configuration. Versions are immutable, so to add, remove, or change group components, you must create `DefinitionVersion` objects that contain new or updated entities.

You can associate new `DefinitionVersion` objects with new or existing `Definition` objects. For example, you can use the `CreateFunctionDefinition` action to create a `FunctionDefinition` that includes the `FunctionDefinitionVersion` as an initial version, or you can use the `CreateFunctionDefinitionVersion` action and reference an existing `FunctionDefinition`.

After you create your group components, you create a `GroupVersion` that contains all `DefinitionVersion` objects that you want to include in the group. Then, you deploy the `GroupVersion`.

To deploy a `GroupVersion`, it must reference a `CoreDefinitionVersion` that contains exactly one `Core`. All referenced entities must be members of the group. Also, a Greengrass service role (p. 443) must be associated with your AWS account in the AWS Region where you are deploying the `GroupVersion`.

**Note**

The `Update` actions in the API are used to change the name of a `Group` or component `Definition` object.

**Updating entities that reference AWS resources**

Greengrass Lambda functions and secret resources (p. 263) define Greengrass-specific properties and also reference corresponding AWS resources. To update these entities, you might make changes to the corresponding AWS resource instead of your Greengrass objects. For example, Lambda functions reference a function in AWS Lambda and also define lifecycle and other properties that are specific to the Greengrass group.
To update Lambda function code or packaged dependencies, make your changes in AWS Lambda. During the next group deployment, these changes are retrieved from AWS Lambda and copied to your local environment.

To update Greengrass-specific properties (p. 178), you create a `FunctionDefinitionVersion` that contains the updated `Function` properties.

**Note**
Greengrass Lambda functions can reference a Lambda function by alias or version ARN. If you reference the alias ARN (recommended), you don't need to update your `FunctionDefinitionVersion` (or `SubscriptionDefinitionVersion`) when you create a new function version in AWS Lambda. For more information, see the section called “Function Aliases and Versions” (p. 177).

**See Also**
- the section called “Get Deployment Notifications” (p. 160)
- the section called “Reset Deployments” (p. 163)
- the section called “Create Bulk Deployments” (p. 165)
- Troubleshooting Deployment Issues (p. 524)

**Get Deployment Notifications**

Using Amazon EventBridge event rules, you can get notifications about state changes for your Greengrass group deployments. EventBridge delivers a near real-time stream of system events that describes changes in AWS resources.

AWS IoT Greengrass emits an event when group deployments change state. 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 triggers 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:

- Trigger post-deployment operations, such as downloading assets and notifying personnel.
- Send notifications upon a successful or failed deployment.
- Publish custom metrics about deployment events.

AWS IoT Greengrass emits an event when a deployment enters the following states: Building, InProgress, Success, and Failure.

**Note**
Monitoring the status of a bulk deployment (p. 165) operation is not currently supported. However, AWS IoT Greengrass emits state-change events for individual group deployments that are part of a bulk deployment.

**Group Deployment Status Change Event**

The event for a deployment state change uses the following format:

```json
{
  "version": "0",
```
You can create rules that apply to one or more groups. You can filter rules by one or more of the following deployment types and deployment states:

**Deployment types**

- **NewDeployment**. The first deployment of a group version.
- **ReDeployment**. A redeployment of a group version.
- **ResetDeployment**. Deletes deployment information stored in the AWS Cloud and on the AWS IoT Greengrass core. For more information, see the section called “Reset Deployments” (p. 163).
- **ForceResetDeployment**. Deletes deployment information stored in the AWS Cloud and reports success without waiting for the core to respond. Also deletes deployment information stored on the core if the core is connected or when it next connects.

**Deployment states**

- **Building**. AWS IoT Greengrass is validating the group configuration and building deployment artifacts.
- **InProgress**. The deployment is in progress on the AWS IoT Greengrass core.
- **Success**. The deployment was successful.
- **Failure**. The deployment failed.

It's possible that events might be duplicated or out of order. To determine the order of events, use the `time` property.

**Note**

AWS IoT Greengrass doesn't use the `resources` property, so it's always empty.

---

**Prerequisites for Creating 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 target, including:
  - Amazon SNS topics
  - AWS Lambda functions
  - Kinesis streams
  - Amazon SQS queues

For more information, see What Is Amazon EventBridge? and Getting Started with Amazon EventBridge in the Amazon EventBridge User Guide.
Configure Deployment 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 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, choose Greengrass Deployment Status Change.
      Note
      The AWS API Call via CloudTrail event type is based on AWS IoT Greengrass integration with AWS CloudTrail. You can use this option to create rules triggered by read or write calls to the AWS IoT Greengrass API. For more information, see the section called “Logging AWS IoT Greengrass API Calls with AWS CloudTrail” (p. 462).
   f. Choose the deployment states that trigger a notification.
      • To receive notifications for all state change events, choose Any state.
      • To receive notifications for some state change events only, choose Specific state(s), and then choose the target states.
   g. Choose the deployment types that trigger a notification.
      • To receive notifications for all deployment types, choose Any state.
      • To receive notifications for some deployment types only, choose Specific state(s), and then choose the target deployment types.
4. Under Select event bus, keep the default event bus options.
5. Under Select targets, 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 Add target.
6. Under Tags - optional, define tags for the rule or leave the fields empty.
7. Choose Create.

Configure Deployment Notifications (CLI)

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.

1. Create the rule.
   • Replace group-id with the ID of your AWS IoT Greengrass group.
Configure Deployment Notifications (AWS CloudFormation)

Use AWS 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

- Deploy AWS IoT Greengrass Groups (p. 154)
- What Is Amazon EventBridge? in the Amazon EventBridge User Guide

Reset Deployments

This feature is available for AWS IoT Greengrass Core v1.1 and later.

You might want to reset a group's deployments to:

- Delete the group (for example, when the group's core has been reimaged.)
- Move the group's core to a different group.
- Revert the group to its state before any deployments.
- Remove the deployment configuration from the core device.
- Delete sensitive data from the core device or from the cloud.
- Deploy a new group configuration to a core without having to replace the core with another in the current group.

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.

```bash
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.
Note
Reset deployments functionality is not available in AWS IoT Greengrass Core Software v1.0.0. You cannot delete a group that has been deployed using v1.0.0.

The reset deployments operation first cleans up all deployment information stored in the cloud for a given group. It then instructs the group's core device to clean up all of its deployment related information as well (Lambda functions, user logs, shadow database and server certificate, but not the user-defined config.json or the Greengrass core certificates). You cannot initiate a reset of deployments for a group if the group currently has a deployment with status of In Progress or Building.

Reset Deployments from the AWS IoT console

You can reset group deployments from group configuration page in the AWS IoT console.

1. In the AWS IoT console, choose Greengrass, and then choose Groups.
2. Choose the target group.
3. From Actions, choose Reset Deployments.

Reset Deployments with the AWS IoT Greengrass API

You can use the ResetDeployments action in the AWS CLI, AWS IoT Greengrass API, or AWS SDK to reset deployments. The examples in this topic use the CLI.

```bash
aws greengrass reset-deployments --group-id GroupId [--force]
```

Arguments for the reset-deployments CLI command:

--group-id

The group ID. Use the list-groups command to get this value.

--force

Optional. Use this parameter if the group's core device has been lost, stolen, or destroyed. This option causes the reset deployment process to report success after all deployment information in the cloud has been cleaned up, without waiting for a core device to respond. However, if the core device is or becomes active, it also performs cleanup operations.

The output of the reset-deployments CLI command looks like this:

```json
{}
```
You can check the status of the reset deployment with the `get-deployment-status` CLI command:

```bash
aws greengrass get-deployment-status --deployment-id DeploymentId --group-id GroupId
```

**Arguments for the `get-deployment-status` CLI command:**

`--deployment-id`  
The deployment ID.

`--group-id`  
The group ID.

The output of the `get-deployment-status` CLI command looks like this:

```
{
  "DeploymentStatus": "Success",
  "UpdatedAt": "2017-04-04T00:00:00.000Z"
}
```

The `DeploymentStatus` is set to `Building` when the reset deployment is being prepared. When the reset deployment is ready but the AWS IoT Greengrass core has not picked up the reset deployment, the `DeploymentStatus` is `InProgress`.

If the reset operation fails, error information is returned in the response.

**See Also**

- *Deploy AWS IoT Greengrass Groups* (p. 154)
- *ResetDeployments* in the *AWS IoT Greengrass API Reference*
- *GetDeploymentStatus* in the *AWS IoT Greengrass API Reference*

## Create Bulk Deployments for Groups

You can use simple API calls to deploy large numbers of Greengrass groups at once. These deployments are triggered with an adaptive rate that has a fixed upper limit.

This tutorial describes how to use the AWS CLI to create and monitor a bulk group deployment in AWS IoT Greengrass. The bulk deployment example in this tutorial contains multiple groups. You can use the example in your implementation to add as many groups as you need.

The tutorial contains the following high-level steps:

1. Create and Upload the Bulk Deployment Input File (p. 166)
2. Create and Configure an IAM Execution Role (p. 167)
3. Allow Your Execution Role Access to Your S3 Bucket (p. 169)
4. Deploy the Groups (p. 170)
5. Test the Deployment (p. 171)
Prerequisites

To complete this tutorial, you need:

- One or more deployable Greengrass groups. For more information about creating AWS IoT Greengrass groups and cores, see Getting Started with AWS IoT Greengrass (p. 64).
- The AWS CLI installed and configured on your machine. For information, see the AWS CLI User Guide.
- An S3 bucket created in the same AWS Region as AWS IoT Greengrass. For information, see Creating and Configuring an S3 Bucket.

Note
Currently, SSE KMS enabled buckets are not supported.

Step 1: Create and Upload the Bulk Deployment Input File

In this step, you create a deployment input file and upload it to your Amazon S3 bucket. This file is a serialized, line-delimited JSON file that contains information about each group in your bulk deployment. AWS IoT Greengrass uses this information to deploy each group on your behalf when you initialize your bulk group deployment.

1. Run the following command to get the groupId for each group you want to deploy. You enter the groupId into your bulk deployment input file so that AWS IoT Greengrass can identify each group to be deployed.

   Note
   You can also find these values in the AWS IoT console. The group ID is shown on the group's Settings page. Group version IDs are shown on the group's Deployments page.

   ```bash
   aws greengrass list-groups
   ```

   The response contains information about each group in your AWS IoT Greengrass account:

   ```json
   {
   "Groups": [
   {
   "Name": "string",
   "Id": "string",
   "Arn": "string",
   "LastUpdatedTimestamp": "string",
   "CreationTimestamp": "string",
   "LatestVersion": "string",
   "LatestVersionArn": "string"
   }
   ],
   "NextToken": "string"
   }
   ```

   Run the following command to get the groupVersionId of each group you want to deploy.

   ```bash
   list-group-versions --group-id groupId
   ```
Step 2: Create and Configure an IAM Execution Role

In this step, you use the IAM console to create a standalone execution role. You then establish a trust relationship between the role and AWS IoT Greengrass and ensure that your IAM user has PassRole privileges for your execution role. This allows AWS IoT Greengrass to assume your execution role and create the deployments on your behalf.

1. Use the following policy to create an execution role. This policy document allows AWS IoT Greengrass to access your bulk deployment input file when it creates each deployment on your behalf.
For more information about creating an IAM role and delegating permissions, see Creating IAM Roles.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "VisualEditor0",
         "Effect": "Allow",
         "Action": ["greengrass:CreateDeployment"],
         "Resource": ["arn:aws:greengrass:region:accountId:/greengrass/groups/groupId1",
                       "arn:aws:greengrass:region:accountId:/greengrass/groups/groupId2",
                       "arn:aws:greengrass:region:accountId:/greengrass/groups/groupId3",
                       "*"
                      ]
      }
   ]
}
```

**Note**
This policy must have a resource for each group or group version in your bulk deployment input file to be deployed by AWS IoT Greengrass. To allow access to all groups, for Resource, specify an asterisk:

```
"Resource": ["*"]
```

2. Modify the trust relationship for your execution role to include AWS IoT Greengrass. This allows AWS IoT Greengrass to use your execution role and the permissions attached to it. For information, see Editing the Trust Relationship for an Existing Role.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "",
         "Effect": "Allow",
         "Principal": {
            "Service": ["greengrass.amazonaws.com"]
         },
         "Action": ["sts:AssumeRole"]
      }
   ]
}
```

3. Give IAM PassRole permissions for your execution role to your IAM user. This IAM user is the one used to initiate the bulk deployment. PassRole permissions allow your IAM user to pass your execution role to AWS IoT Greengrass for use. For more information, see Granting a User Permissions to Pass a Role to an AWS Service.

Use the following example to update your trust policy document. Modify this example, as necessary.

```
{
   "Version": "2012-10-17",
   "Statement": [
```
Step 3: Allow Your Execution Role Access to Your S3 Bucket

To start your bulk deployment, your execution role must be able to read your bulk deployment input file from your Amazon S3 bucket. Attach the following example policy to your Amazon S3 bucket so its GetObject permissions are accessible to your execution role.

For more information, see How Do I Add an S3 Bucket Policy?

You can use the following command in your terminal to check your bucket's policy:

```
aws s3api get-bucket-policy --bucket my-bucket
```

**Note**
You can directly modify your execution role to grant it permission to your Amazon S3 bucket's GetObject permissions instead. To do this, attach the following example policy to your execution role.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt1508193814000",
      "Effect": "Allow",
      "Action": [
        "iam:PassRole"
      ],
      "Resource": [
        "arn:aws:iam::123456789012:user/executionRoleArn"
      ]
    }
  ]
}
```
Step 4: Deploy the Groups

In this step, you start a bulk deployment operation for all group versions configured in your bulk deployment input file. The deployment action for each of your group versions is of type NewDeploymentType.

Note
You cannot call StartBulkDeployment while another bulk deployment from the same account is still running. The request is rejected.

1. Use the following command to start the bulk deployment.

   We recommend that you include an X-Amzn-Client-Token token in every StartBulkDeployment request. These requests are idempotent with respect to the token and the request parameters. This token can be any unique, case-sensitive string of up to 64 ASCII characters.

   ```
   aws greengrass start-bulk-deployment --cli-input-json "{
   "InputFileUri": "URI of file in S3 bucket",
   "ExecutionRoleArn": "ARN of execution role",
   "AmznClientToken": "your Amazon client token"
   }"
   ```

   The command should result in a successful status code of 200, along with the following response:

   ```
   {
   "bulkDeploymentId": "UUID"
   }
   ```

   Make a note of the bulk deployment ID. It can be used to check the status of your bulk deployment.

   Note
   Although bulk deployment operations are not currently supported, you can create Amazon EventBridge event rules to get notifications about deployment status changes for individual groups. For more information, see the section called “Get Deployment Notifications” (p. 160).

2. Use the following command to check the status of your bulk deployment.

   ```
   aws greengrass get-bulk-deployment-status --bulk-deployment-id 1234567
   ```

   The command should return a successful status code of 200 in addition to a JSON payload of information:

   ```
   {
   "BulkDeploymentStatus": "Running",
   `}
BulkDeploymentStatus contains the current status of the bulk execution. The execution can have one of six different statuses:

- **Initializing.** The bulk deployment request has been received, and the execution is preparing to start.
- **Running.** The bulk deployment execution has started.
- **Completed.** The bulk deployment execution has finished processing all records.
- **Stopping.** The bulk deployment execution has received a command to stop and will terminate shortly. You can't start a new bulk deployment while a previous deployment is in the Stopping state.
- **Stopped.** The bulk deployment execution has been manually stopped.
- **Failed.** The bulk deployment execution has encountered an error and terminated. You can find error details in the ErrorDetails field.

The JSON payload also includes statistical information about the progress of the bulk deployment. You can use this information to determine how many groups have been processed and how many have failed. The statistical information includes:

- **RecordsProcessed:** The number of group records that were attempted.
- **InvalidInputRecords:** The total number of records that returned a non-retryable error. For example, this can occur if a group record from the input file uses an invalid format or specifies a nonexistent group version, or if the execution doesn't grant permission to deploy a group or group version.
- **RetryAttempts:** The number of deployment attempts that returned a retryable error. For example, a retry is triggered if the attempt to deploy a group returns a throttling error. A group deployment can be retried up to five times.

In the case of a bulk deployment execution failure, this payload also includes an ErrorDetails section that can be used for troubleshooting. It contains information about the cause of the execution failure.

You can periodically check the status of the bulk deployment to confirm that it is progressing as expected. After the deployment is complete, RecordsProcessed should be equal to the number of deployment groups in your bulk deployment input file. This indicates that each record has been processed.

### Step 5: Test the Deployment

Use the **ListBulkDeployments** command to find the ID of your bulk deployment.
aws greengrass list-bulk-deployments

This command returns a list of all of your bulk deployments from most to least recent, including your BulkDeploymentId.

```json
{
    "BulkDeployments": [
    {
        "BulkDeploymentId": 1234567,
        "BulkDeploymentArn": "string",
        "CreatedAt": "string"
    }
    ],
    "NextToken": "string"
}
```

Now call the `ListBulkDeploymentDetailedReports` command to gather detailed information about each deployment.

```bash
aws greengrass list-bulk-deployment-detailed-reports --bulk-deployment-id 1234567
```

The command should return a successful status code of 200 along with a JSON payload of information:

```json
{
    "BulkDeploymentResults": [
    {
        "DeploymentId": "string",
        "GroupVersionedArn": "string",
        "CreatedAt": "string",
        "DeploymentStatus": "string",
        "ErrorMessage": "string",
        "ErrorDetails": [
        {
            "DetailedErrorCode": "string",
            "DetailedErrorMessage": "string"
        }
        ],
    },
    "NextToken": "string"
}
```

This payload usually contains a paginated list of each deployment and its deployment status from most to least recent. It also contains more information in the event of a bulk deployment execution failure. Again, the total number of deployments listed should be equal to the number of groups you identified in your bulk deployment input file.

The information returned can change until the deployments are in a terminal state (success or failure). You can call this command periodically until then.
Troubleshooting Bulk Deployments

If the bulk deployment is not successful, you can try the following troubleshooting steps. Run the commands in your terminal.

Troubleshoot input file errors

The bulk deployment can fail in the event of syntax errors in the bulk deployment input file. This returns a bulk deployment status of Failed with an error message indicating the line number of the first validation error. There are four possible errors:

* InvalidInputFile: Missing GroupId at line number: line number

This error indicates that the given input file line is unable to register the specified parameter. The possible missing parameters are the GroupId and the GroupVersionId.

* InvalidInputFile: Invalid deployment type at line number : line number. Only valid type is 'NewDeployment'.

This error indicates that the given input file line lists an invalid deployment type. At this time, the only supported deployment type is a NewDeployment.

* Line %s is too long in S3 File. Valid line is less than 256 chars.

This error indicates that the given input file line is too long and must be shortened.

* Failed to parse input file at line number: line number

This error indicates that the given input file line is not considered valid json.

Check for concurrent bulk deployments

You cannot start a new bulk deployment while another one is still running or in a non-terminal state. This can result in a Concurrent Deployment Error. You can use the ListBulkDeployments command to verify that a bulk deployment is not currently running. This command lists your bulk deployments from most to least recent.

```
{
  "BulkDeployments": [
    {
      "BulkDeploymentId": BulkDeploymentId,
      "BulkDeploymentArn": "string",
      "CreatedAt": "string"
    }
  ],
  "NextToken": "string"
}
```

Use the BulkDeploymentId of the first listed bulk deployment to run the GetBulkDeploymentStatus command. If your most recent bulk deployment is in a running state (Initializing or Running), use the following command to stop the bulk deployment.
This action results in a status of Stopping until the deployment is Stopped. After the deployment has reached a Stopped status, you can start a new bulk deployment.

**Check ErrorDetails**

Run the `GetBulkDeploymentStatus` command to return a JSON payload that contains information about any bulk deployment execution failure.

```
"Message": "string",
"ErrorDetails": [
  {
    "DetailedErrorCode": "string",
    "DetailedErrorMessage": "string"
  }
]
```

When exiting with an error, the `ErrorDetails` JSON payload that is returned by this call contains more information about the bulk deployment execution failure. An error status code in the 400 series, for example, indicates an input error, either in the input parameters or the caller dependencies.

**Check the AWS IoT Greengrass core log**

You can troubleshoot issues by viewing the AWS IoT Greengrass core logs. Use the following commands to view `runtime.log`:

```bash
cd /greengrass/ggc/var/log
sudo cat system/runtime.log | more
```

For more information about AWS IoT Greengrass logging, see Monitoring with AWS IoT Greengrass Logs (p. 457).

**See Also**

For more information, see the following resources:

- AWS IoT Greengrass CLI Reference
- AWS IoT Greengrass API Reference
Run Lambda Functions on the AWS IoT Greengrass Core

AWS IoT Greengrass provides a containerized Lambda runtime environment for user-defined code that you author in AWS Lambda. Lambda functions that are deployed to an AWS IoT Greengrass core run in the core’s local Lambda runtime. Local Lambda functions can be triggered by local events, messages from the cloud, and other sources, which brings local compute functionality to connected devices. For example, you can use Greengrass Lambda functions to filter device data before transmitting the data to the cloud.

To deploy a Lambda function to a core, you add the function to a Greengrass group (by referencing the existing Lambda function), configure group-specific settings for the function, and then deploy the group. If the function accesses AWS services, you also must add any required permissions to the group role.

You can configure parameters that determine how the Lambda functions run, including permissions, isolation, memory limits, and more. For more information, see the section called “Controlling Greengrass Lambda Function Execution” (p. 178).

**Note**

These settings also make it possible to run AWS IoT Greengrass in a Docker container. For more information, see the section called “Run AWS IoT Greengrass in a Docker Container” (p. 189).

The following table lists supported AWS Lambda runtimes and the versions of AWS IoT Greengrass Core software that they can run on.

<table>
<thead>
<tr>
<th>Language or platform</th>
<th>GGC version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python 2.7</td>
<td>1.0 or later</td>
</tr>
<tr>
<td>Python 3.7</td>
<td>1.9</td>
</tr>
<tr>
<td>Java 8</td>
<td>1.1 or later</td>
</tr>
<tr>
<td>Node.js 6.10</td>
<td>1.1 or later</td>
</tr>
<tr>
<td>Node.js 8.10</td>
<td>1.9</td>
</tr>
<tr>
<td>C, C++</td>
<td>1.6 or later</td>
</tr>
</tbody>
</table>

**SDKs for Greengrass Lambda Functions**

AWS provides three SDKs that can be used by Greengrass Lambda functions running on an AWS IoT Greengrass core. These SDKs are contained in different packages, so functions can use them simultaneously. To use an SDK in a Greengrass Lambda function, include it in your deployment package.

**AWS IoT Greengrass Core SDK**

Enables local Lambda functions to interact with the core to:

- Exchange MQTT messages with AWS IoT.
- Exchange MQTT messages with connectors, devices, and other Lambda functions in the Greengrass group.
• Interact with the local shadow service.
• Invoke other local Lambda functions.
• Access secret resources (p. 263).

AWS IoT Greengrass provides the AWS IoT Greengrass Core SDK in the following languages and platforms on GitHub.

• AWS IoT Greengrass Core SDK for Java
• AWS IoT Greengrass Core SDK for Node.js
• AWS IoT Greengrass Core SDK for Python
• AWS IoT Greengrass Core SDK for C

If you're running Python Lambda functions, you can also use pip to install the AWS IoT Greengrass Core SDK for Python on the core device. Then you can deploy your functions without including the SDK in the Lambda function deployment package. For more information, see greengrasssdk.

**Note**
To use pip to install the Python SDK, run the following command in your core device terminal.

```
pip install greengrasssdk
```

### AWS IoT Greengrass Machine Learning SDK

Enables local Lambda functions to consume machine learning (ML) models that are deployed to the Greengrass core as ML resources. Lambda functions can use the SDK to invoke and interact with a local inference service that's deployed to the core as a connector. Lambda functions and ML connectors can also use the SDK to send data to the ML Feedback connector for uploading and publishing. For more information, including code examples that use the SDK, see the section called "ML Image Classification" (p. 326), the section called "ML Object Detection" (p. 343), and the section called "ML Feedback" (p. 315).

The following table lists supported languages or platforms for SDK versions and the versions of AWS IoT Greengrass Core software they can run on.

<table>
<thead>
<tr>
<th>SDK version</th>
<th>Language or platform</th>
<th>Required GGC version</th>
<th>Changelog</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.0</td>
<td>Python 3.7 or 2.7</td>
<td>1.9.3 or later</td>
<td>Added Python 3.7 support and new feedback client.</td>
</tr>
<tr>
<td>1.0.0</td>
<td>Python 2.7</td>
<td>1.7 or later</td>
<td>Initial release.</td>
</tr>
</tbody>
</table>

For download information, see the section called "AWS IoT Greengrass ML SDK Software" (p. 23).

### AWS SDKs

Enables local Lambda functions to make direct calls to AWS services, such as Amazon S3, DynamoDB, AWS IoT, and AWS IoT Greengrass. To use an AWS SDK in a Greengrass Lambda function, you must include it in your deployment package. When you use the AWS SDK in the same package
as the AWS IoT Greengrass Core SDK, make sure that your Lambda functions use the correct namespaces. Greengrass Lambda functions can't communicate with cloud services when the core is offline.

Download the AWS SDKs from the Getting Started Resource Center.

For more information about creating a deployment package, see the section called “Create and Package a Lambda Function” (p. 84) in the Getting Started tutorial or Creating a Deployment Package in the AWS Lambda Developer Guide.

Migrating Cloud-Based Lambda Functions

The AWS IoT Greengrass Core SDK follows the AWS SDK programming model, which makes it easy to port Lambda functions that are developed for the cloud to Lambda functions that run on an AWS IoT Greengrass core.

For example, the following Python Lambda function uses the AWS SDK for Python to publish a message to the topic some/topic in the cloud:

```python
import boto3
client = boto3.client('iot-data')
response = client.publish(
    topic = 'some/topic',
    qos = 0,
    payload = 'Some payload'.encode()
)
```

To port the function for an AWS IoT Greengrass core, in the `import` statement and `client` initialization, change the `boto3` module name to `greengrasssdk`, as shown in the following example:

```python
import greengrasssdk
client = greengrasssdk.client('iot-data')
response = client.publish(
    topic = 'some/topic',
    qos = 0,
    payload = 'Some payload'.encode()
)
```

Note
The AWS IoT Greengrass Core SDK supports sending MQTT messages with QoS = 0 only.

The similarity between programming models also makes it possible for you to develop your Lambda functions in the cloud and then migrate them to AWS IoT Greengrass with minimal effort. Lambda executables (p. 187) don't run in the cloud, so you can't use the AWS SDK to develop them in the cloud before deployment.

Reference Lambda Functions by Alias or Version

Greengrass groups can reference a Lambda function by alias (recommended) or by version. Using an alias makes it easier to manage code updates because you don't have to change your subscription table or group definition when the function code is updated. Instead, you just point the alias to the new function version. Aliases resolve to version numbers during group deployment. When you use aliases, the resolved version is updated to the version that the alias is pointing to at the time of deployment.
AWS IoT Greengrass doesn't support Lambda aliases for $LATEST versions. $LATEST versions aren't bound to immutable, published function versions and can be changed at any time, which is counter to the AWS IoT Greengrass principle of version immutability.

A common practice for keeping your Greengrass Lambda functions updated with code changes is to use an alias named PRODUCTION in your Greengrass group and subscriptions. As you promote new versions of your Lambda function into production, point the alias to the latest stable version and then redeploy the group. You can also use this method to roll back to a previous version.

Controlling Execution of Greengrass Lambda Functions by Using Group-Specific Configuration

AWS IoT Greengrass provides cloud-based management of Greengrass Lambda functions. You can configure details of how the Lambda function behaves when it runs in a group. Although a function's code and dependencies are managed using AWS Lambda, AWS IoT Greengrass supports the following group-specific configuration settings:

Run as

The access identity used to run each Lambda function. By default, Lambda functions run as the group's default access identity (p. 183). Typically, this is the standard AWS IoT Greengrass system accounts (ggc_user and ggc_group). You can change the setting and choose the user ID and group ID that have the permissions required to run the Lambda function. You can override both UID and GID or just one if you leave the other field blank. This setting gives you more granular control over access to device resources. We recommend that you configure your Greengrass hardware with appropriate resource limits, file permissions, and disk quotas for the users and groups whose permissions are used to run Lambda functions.

This feature is available for AWS IoT Greengrass Core v1.7 and later.

Important

We recommend that you avoid running as root unless absolutely necessary. When you run a Lambda function as root, you increase the risk of unintended changes, such as accidentally deleting a critical file. In addition, running as root increases the risks to your data and device from malicious individuals. If you do need to run as root, you must update the AWS IoT Greengrass configuration to enable it. For more information, see the section called “Running a Lambda Function as Root” (p. 180).

UID (number)

The user ID for the user that has the permissions required to run the Lambda function. This setting is only available if you choose Run as another user ID/group ID. You can use the getent passwd command on your AWS IoT Greengrass device to look up the user ID you want to use to run the Lambda function.

GID (number)

The group ID for the group that has the permissions required to run the Lambda function. This setting is only available if you choose Run as another user ID/group ID. You can use the getent group command on your AWS IoT Greengrass device to look up the group ID you want to use to run the Lambda function.

Containerization

Choose whether the Lambda function runs with the default containerization for the group, or specify the containerization that should always be used for this Lambda function. To run without enabling your device kernel namespace and cgroup, all your Lambda functions must run without
containerization. You can accomplish this easily by setting the default containerization for the group. For information, see the section called “Setting Default Containerization for Lambda Functions in a Group” (p. 184).

This feature is available for AWS IoT Greengrass Core v1.7 and later.

**Note**
We recommend that you run Lambda functions in a Greengrass container unless your use case requires them to run without containerization. When your Lambda functions are running in a Greengrass container, you can use attached resources, and gain the benefits of isolation and increased security. Before you change the containerization, see the section called “Considerations When Choosing Lambda Function Containerization” (p. 181).

**Memory limit**

The memory allocation for the function. The default is 16 MB.

**Note**
This setting is not available when you run a Lambda function without containerization. Lambda functions run without containerization have no memory limit. The memory limit setting is discarded when you change the Lambda function to run without containerization.

**Timeout**

The amount of time before the function or request is terminated. The default is 3 seconds.

**Lifecycle**

A Lambda function lifecycle can be on-demand or long-lived. The default is on-demand.

An on-demand Lambda function starts in a new or reused container when invoked. Requests to the function might be processed by any available container. A long-lived—or pinned—Lambda function starts automatically after AWS IoT Greengrass starts and keeps running in its own container (or sandbox). All requests to the function are processed by the same container. For more information, see the section called “Lifecycle Configuration” (p. 187).

**Read access to /sys directory**

Whether the function can access the host’s /sys folder. Use this when the function must read device information from /sys. The default is false.

**Note**
This setting is not available when you run a Lambda function without containerization. The value of this setting is discarded when you change the Lambda function to run without containerization.

**Input payload data type**

The expected encoding type of the input payload for the function, either JSON or binary. The default is JSON.

Support for the binary encoding type is available starting in AWS IoT Greengrass Core Software v1.5.0 and AWS IoT Greengrass Core SDK v1.1.0. Accepting binary input data can be useful for functions that interact with device data, because the restricted hardware capabilities of devices often make it difficult or impossible for them to construct a JSON data type.

**Note**
Lambda executables (p. 187) support the binary encoding type only, not JSON.

**Environment variables**

Key-value pairs that can dynamically pass settings to function code and libraries. Local environment variables work the same way as AWS Lambda function environment variables, but are available in the core environment.
Resource access policies

A list of up to 10 local resources (p. 200), secret resources (p. 263), and machine learning resources (p. 221) that the Lambda function is allowed to access, and the corresponding read-only or read-write permission. In the console, these affiliated resources are listed on the function's Resources page.

Note
Resource access policies apply only when Lambda functions are run in a Greengrass container. When you run Lambda functions without containerization, you can directly access local resources. You can also access secret resources when running without containerization.

Running a Lambda Function as Root

This feature is available for AWS IoT Greengrass Core v1.7 and later.

Before you can run one or more Lambda functions as root, you must first update the AWS IoT Greengrass configuration to enable support. Support for running Lambda functions as root is off by default. The deployment fails if you try to deploy a Lambda function and run it as root (UID and GID of 0) and you haven’t updated the AWS IoT Greengrass configuration. An error like the following appears in the runtime log (greengrass_root/ggc/var/log/system/runtime.log):

```
lambda(s) [list of function arns] are configured to run as root while Greengrass is not configured to run lambdas with root permissions
```

Important
We recommend that you avoid running as root unless absolutely necessary. When you run a Lambda function as root, you increase the risk of unintended changes, such as accidentally deleting a critical file. In addition, running as root increases the risks to your data and device from malicious individuals.

To allow Lambda functions to run as root

1. On your AWS IoT Greengrass device, navigate to the greengrass-root/config folder.
   
   Note
   By default, greengrass-root is the /greengrass directory.

2. Edit the config.json file to add "allowFunctionsToRunAsRoot" : "yes" to the runtime field. For example:

```
{
  "coreThing" : {
    "..."
  },
  "runtime" : {
    "..."
    "allowFunctionsToRunAsRoot" : "yes"
  },
  "..."
}
```

3. Use the following commands to restart AWS IoT Greengrass:

```
cd /greengrass/ggc/core
sudo ./greengrassd restart
```
Now you can set the user ID and group ID (UID/GID) of Lambda functions to 0 to run that Lambda function as root.

You can change the value of "allowFunctionsToRunAsRoot" to "no" and restart AWS IoT Greengrass if you want to disallow Lambda functions to run as root.

Considerations When Choosing Lambda Function Containerization

This feature is available for AWS IoT Greengrass Core v1.7 and later.

By default, Lambda functions run inside an AWS IoT Greengrass container. That container provides isolation between your functions and the host, which offers more 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 restricting access to resources.

Here are some example use cases for running without containerization:

• You want to run AWS IoT Greengrass on a device that does not support container mode (for example, because you are using a special Linux distribution or have a kernel version that is too old).
• 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, such as pluggable devices.
• You have a legacy application that was written as a process and you have encountered issues when running it as a containerized Lambda function.

Containerization Differences

<table>
<thead>
<tr>
<th>Containerization</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS IoT Greengrass container</td>
<td>• All AWS IoT Greengrass features are available when you run a Lambda function in a Greengrass container.</td>
</tr>
<tr>
<td></td>
<td>• Lambda functions that run in a Greengrass container do not have access to the deployed code of other Lambda functions, even if they run with the same group ID. In other words, your Lambda functions run with greater isolation from one another.</td>
</tr>
<tr>
<td></td>
<td>• Because Lambda functions that run in an AWS IoT Greengrass container have all child processes execute in the same container as the Lambda function, the child processes are terminated when the Lambda function is terminated.</td>
</tr>
<tr>
<td>No container</td>
<td>• Not all AWS IoT Greengrass features are available when you run a Lambda function without containerization. You must access the file system directly instead of using attached</td>
</tr>
</tbody>
</table>
### Considerations When Choosing Lambda Function Containerization

<table>
<thead>
<tr>
<th>Containerization</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>local device and volume resources (p. 200). Currently, machine learning (ML) model resources (p. 221) aren't supported without containerization.</td>
</tr>
<tr>
<td></td>
<td>• The Lambda function has read-only access to the deployed code of other Lambda functions that are running with the same group ID.</td>
</tr>
<tr>
<td></td>
<td>• Lambda functions that spawn child processes in a different process session or with an overridden SIGHUP (signal hangup) handler, such as with the nohup utility, are not automatically terminated by AWS IoT Greengrass when the parent Lambda function is terminated.</td>
</tr>
<tr>
<td></td>
<td>• Connectors (p. 283), local device and volume resources, Lambda function memory limits, and ML model resources aren’t supported when the default Lambda function containerization for the Greengrass group is set to No container. These features require AWS IoT Greengrass to run with containerization.</td>
</tr>
</tbody>
</table>

Changing the containerization for a Lambda function can cause problems when you deploy it. If you had assigned local resources to your Lambda function that are no longer available with your new containerization settings, deployment fails.

- When you change a Lambda function from running in a Greengrass container to running without containerization, memory limits for the function are discarded. You must access the file system directly instead of using attached local resources. You must remove any attached resources before you deploy.
- 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. You can configure those settings for each Lambda function before you deploy.

### To change containerization settings for a Lambda function

1. In the AWS IoT console, choose **Greengrass**, and then choose **Groups**.
2. Choose the group that contains the Lambda function whose settings you want to change.
3. Choose **Lambdas**.
4. On the Lambda function that you want to change, choose the ellipsis (...) and then choose **Edit configuration**.
5. Change the containerization settings. If you configure the Lambda function to run in a Greengrass container, you must also set Memory limit and Read access to /sys directory.
6. Choose **Update** to save the changes to your Lambda function.

The changes take effect when the group is deployed.

You can also use the CreateFunctionDefinition and CreateFunctionDefinitionVersion in the AWS IoT Greengrass API Reference. If you are changing the containerization setting, be sure to update the other parameters too. For example, if you are changing from running a Lambda function in a Greengrass container to running without containerization, be sure to clear the MemorySize parameter.
Determine the Isolation Modes Supported by Your Greengrass Device

You can use the AWS IoT Greengrass dependency checker to determine which isolation modes (Greengrass container/no container) are supported by your Greengrass device.

To run the AWS IoT Greengrass dependency checker

1. Download and run the AWS IoT Greengrass dependency checker from the GitHub repository.

   unzip greengrass-dependency-checker-GGCv1.9.x.zip
   cd greengrass-dependency-checker-GGCv1.9.x
   sudo modprobe configs
   sudo ./check_ggc_dependencies | more

2. Where `more` appears, press the Spacebar key to display another page of text.

   For information about the `modprobe` command, run `man modprobe` in the terminal.

Setting the Default Access Identity for Lambda Functions in a Group

This feature is available for AWS IoT Greengrass Core v1.8 and later.

For more control over access to device resources, you can configure the default access identity used to run Lambda functions in the group. This setting determines the default permissions given to your Lambda functions when they run on the core device. To override the setting for individual functions in the group, you can use the function’s `Run as` property. For more information, see Run as (p. 178).

This group-level setting is also used for running the underlying AWS IoT Greengrass Core software. This consists of system Lambda functions that manage operations, such as message routing, local shadow sync, and automatic IP address detection.

The default access identity can be configured to run as the standard AWS IoT Greengrass system accounts (ggc_user and ggc_group) or use the permissions of another user or group. We recommend that you configure your Greengrass hardware with appropriate resource limits, file permissions, and disk quotas for any users and groups whose permissions are used to run user-defined or system Lambda functions.

To modify the default access identity for your AWS IoT Greengrass group

1. In the AWS IoT console, choose Greengrass, and then choose Groups.
2. Choose the group whose settings you want to change.
3. Choose Settings.
4. Under Lambda runtime environment, for Default Lambda function user ID/group ID, choose Another user ID/group ID.

   When you choose this option, the **UID (number)** and **GID (number)** fields are displayed.

5. Enter a user ID, group ID, or both. If you leave a field blank, the respective Greengrass system account (ggc_user or ggc_group) is used.

   - **For UID (number)**, enter the user ID for the user who has the permissions you want to use by default to run Lambda functions in the group. You can use the `getent passwd` command on your AWS IoT Greengrass device to look up the user ID.
• For **GID (number)**, enter the group ID for the group that has the permissions you want to use by default to run Lambda functions in the group. You can use the `getent group` command on your AWS IoT Greengrass device to look up the group ID.

**Important**
Running as the root user increases risks to your data and device. Do not run as root (UID/GID=0) unless your business case requires it. For more information, see the section called “Running a Lambda Function as Root” (p. 180).

The changes take effect when the group is deployed.

**Setting Default Containerization for Lambda Functions in a Group**

This feature is available for AWS IoT Greengrass Core v1.7 and later.

You can modify the group settings to specify the default containerization for Lambda functions in the group. You can override this setting for one or more Lambda functions in the group if you want the Lambda functions to run with containerization different from the group default. Before you change containerization settings, see the section called “Considerations When Choosing Lambda Function Containerization” (p. 181).

**Important**
If you want to change the default containerization for the group, but have one or more functions that use a different containerization, change the settings for the Lambda functions before you change the group setting. If you change the group containerization setting first, the values for the **Memory limit** and **Read access to /sys directory** settings are discarded.

To modify containerization settings for your AWS IoT Greengrass group

1. In the AWS IoT console, choose **Greengrass**, and then choose **Groups**.
2. Choose the group whose settings you want to change.
3. Choose **Settings**.
4. Under **Lambda runtime environment**, change the containerization setting.

The changes take effect when the group is deployed.

**Communication Flows for Greengrass Lambda Functions**

Greengrass Lambda functions support several methods of communicating with other members of the AWS IoT Greengrass group, local services, and cloud services (including AWS services).

**Communication Using MQTT Messages**

Lambda functions can send and receive MQTT messages using a publish-subscribe pattern that's controlled by subscriptions.

This communication flow allows Lambda functions to exchange messages with the following entities:
• Devices in the group.
• Connectors in the group.
• Other Lambda functions in the group.
• AWS IoT.
• Local Device Shadow service.

A subscription defines a message source, a message target, and a topic (or subject) that's used to route messages from the source to the target. Messages that are published to a Lambda function are passed to the function's registered handler. Subscriptions enable more security and provide predictable interactions. For more information, see the section called “Greengrass Messaging Workflow” (p. 441).

Note
When the core is offline, Greengrass Lambda functions can exchange messages with devices, connectors, other functions, and local shadows, but messages to AWS IoT are queued. For more information, see the section called “MQTT Message Queue” (p. 55).

Other Communication Flows

• To interact with local resources and machine learning models on a core device, Greengrass Lambda functions use platform-specific operating system interfaces. For example, you can use the open method in the os module in Python 2.7 functions. To allow a function to access a resource, the function must be affiliated with the resource and granted read-only or read-write permission. For more information, including AWS IoT Greengrass core version availability, see Access Local Resources (p. 200) and Perform Machine Learning Inference (p. 221).

Note
If you run your Lambda function without containerization, you cannot use attached local resources and must access those resources directly. Machine learning resources are not available for Lambda functions that run without containerization.

• Greengrass Lambda functions can use the AWS SDK to communicate with AWS services. For more information, see AWS SDK (p. 176).

• Greengrass Lambda functions can use third-party interfaces to communicate with external cloud services, similar to cloud-based Lambda functions.

Note
Greengrass Lambda functions can't communicate with AWS or other cloud services when the core is offline.

Retrieve the Input MQTT Topic (or Subject)

AWS IoT Greengrass uses subscriptions to control the exchange of MQTT messages between devices, Lambda functions, and connectors in a group, and with AWS IoT or the local shadow service. Subscriptions define a message source, message target, and an MQTT topic used to route messages. When the target is a Lambda function, the function's handler is invoked when the source publishes a message. For more information, see the section called “Communication Using MQTT Messages” (p. 184).

The following example shows how a Lambda function can get the input topic from the context that's passed to the handler. It does this by accessing the subject key from the context hierarchy (context.client_context.custom['subject']). The example also parses the input JSON message and then publishes the parsed topic and message.
Note
In the AWS IoT Greengrass API, the topic of a subscription is represented by the subject property.

```python
import greengrasssdk
import logging

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

OUTPUT_TOPIC = 'test/topic_results'

def get_input_topic(context):
    try:
        topic = context.client_context.custom['subject']
    except Exception as e:
        logging.error('Topic could not be parsed. ' + repr(e))
    return topic

def get_input_message(event):
    try:
        message = event['test-key']
    except Exception as e:
        logging.error('Message could not be parsed. ' + repr(e))
    return message

def function_handler(event, context):
    try:
        input_topic = get_input_topic(context)
        input_message = get_input_message(event)
        response = 'Invoked on topic "%s" with message "%s"' % (input_topic, input_message)
        logging.info(response)
    except Exception as e:
        logging.error(e)

    client.publish(topic=OUTPUT_TOPIC, payload=response)
    return
```

To test the function, add it to your group using the default configuration settings. Then, add the following subscriptions and deploy the group. For instructions, see the section called "Module 3 (Part 1): Lambda Functions on AWS IoT Greengrass" (p. 84).

After the deployment is completed, invoke the function.

1. In the AWS IoT console, open the **Test** page.
2. Subscribe to the **test/topic_results** topic.
3. Publish a message to the **test/input_message** topic. For this example, you must include the **test-key** property in the JSON message.
"test-key": "Some string value"
}

If successful, the function publishes the input topic and message string to the test/topic_results topic.

Lifecycle Configuration for Greengrass Lambda Functions

The Greengrass Lambda function lifecycle determines when a function starts and how it creates and uses containers. The lifecycle also determines how variables and preprocessing logic that are outside of the function handler are retained.

AWS IoT Greengrass supports the on-demand (default) or long-lived lifecycles:

- **On-demand** functions start when they are invoked and stop when there are no tasks left to execute. An invocation of the function creates a separate container (or sandbox) to process invocations, unless an existing container is available for reuse. Data that's sent to the function might be pulled by any of the containers.

  Multiple invocations of an on-demand function can run in parallel.

  Variables and preprocessing logic that are defined outside of the function handler are not retained when new containers are created.

- **Long-lived** (or pinned) functions start automatically when the AWS IoT Greengrass core starts and run in a single container. All data that's sent to the function is pulled by the same container.

  Multiple invocations are queued until earlier invocations are executed.

  Variables and preprocessing logic that are defined outside of the function handler are retained for every invocation of the handler.

  Long-lived Lambda functions are useful when you need to start doing work without any initial input. For example, a long-lived function can load and start processing an ML model to be ready when the function starts receiving device data.

  **Note**
  
  Remember that long-lived functions have timeouts that are associated with invocations of their handler. If you want to execute indefinitely running code, you must start it outside the handler. Make sure that there's no blocking code outside the handler that might prevent the function from completing its initialization.

  These functions run unless the core stops (for example, during a group deployment or a device reboot) or the function enters an error state (such as a handler timeout, uncaught exception, or when it exceeds its memory limits).

For more information about container reuse, see [Understanding Container Reuse in AWS Lambda](https://aws.amazon.com/blogs/compute/) on the AWS Compute Blog.

Lambda Executables

This feature is available for AWS IoT Greengrass Core v1.6 and later.
A Lambda executable is a type of Greengrass Lambda function that you can use to run binary code in the core environment. It lets you execute device-specific functionality natively and benefit from the smaller footprint of compiled code. Lambda executables can be invoked by events, invoke other functions, and access local resources.

Lambda executables support the binary encoding type only (not JSON), but otherwise you can manage them in your Greengrass group and deploy them like other Greengrass Lambda functions. However, the process of creating Lambda executables is different from creating Python, Java, and Node.js Lambda functions:

- You can't use the AWS Lambda console to create (or manage) a Lambda executable. You can create a Lambda executable only by using the AWS Lambda API.
- You upload the function code to AWS Lambda as a compiled executable that includes the AWS IoT Greengrass Core SDK for C.
- You specify the executable name as the function handler.

Lambda executables must implement certain calls and programming patterns in their function code. For example, the main method must:

- Call `gg_global_init` to initialize Greengrass internal global variables. This function must be called before creating any threads, and before calling any other AWS IoT Greengrass Core SDK functions.
- Call `gg_runtime_start` to register the function handler with the Greengrass Lambda runtime. This function must be called during initialization. Calling this function causes the current thread to be used by the runtime. The optional `GG_RT_OPT_ASYNC` parameter tells this function to not block, but instead to create a new thread for the runtime. This function uses a `SIGTERM` handler.

The following snippet is the main method from the `simple_handler.c` code example on GitHub.

```c
int main() {
    gg_error err = GGE_SUCCESS;
    err = gg_global_init(0);
    if(err) {
        gg_log(GG_LOG_ERROR, "gg_global_init failed %d", err);
        goto cleanup;
    }

    gg_runtime_start(handler, 0);

cleanup:
    return -1;
}
```

For more information about requirements, constraints, and other implementation details, see AWS IoT Greengrass Core SDK for C.

**Create a Lambda Executable**

After you compile your code along with the SDK, use the AWS Lambda API to create a Lambda function and upload your compiled executable.

**Note**
Your function must be compiled with a C89 compatible compiler.

The following example uses the create-function CLI command to create a Lambda executable. The command specifies:
The name of the executable for the handler. This must be the exact name of your compiled executable.

• The path to the .zip file that contains the compiled executable.

• `arn:aws:greengrass:::runtime/function/executable` for the runtime. This is the runtime for all Lambda executables.

**Note**
For `role`, you can specify the ARN of any Lambda execution role. AWS IoT Greengrass doesn't use this role, but the parameter is required to create the function. For more information about Lambda execution roles, see AWS Lambda Permissions Model in the AWS Lambda Developer Guide.

```bash
aws lambda create-function \\
  --region aws-region \\
  --function-name function-name \\
  --handler executable-name \\
  --role role-arn \\
  --zip-file fileb://file-name.zip \\
  --runtime arn:aws:greengrass:::runtime/function/executable
```

Next, use the AWS Lambda API to publish a version and create an alias.

• **Use** `publish-version` **to publish a function version.**

```bash
aws lambda publish-version \\
  --function-name function-name \\
  --region aws-region
```

• **Use** `create-alias` **to create an alias the points to the version you just published.** We recommend that you reference Lambda functions by alias when you add them to a Greengrass group.

```bash
aws lambda create-alias \\
  --function-name function-name \\
  --name alias-name \\
  --function-version version-number \\
  --region aws-region
```

**Note**
The AWS Lambda console doesn't display Lambda executables. To update the function code, you must use the AWS Lambda API.

Then, add the Lambda executable to a Greengrass group, configure it to accept binary input data in its group-specific settings, and deploy the group. You can do this in the AWS IoT Greengrass console or by using the AWS IoT Greengrass API.

## Running AWS IoT Greengrass in a Docker Container

This feature is available for AWS IoT Greengrass Core v1.7 and later.

AWS IoT Greengrass can be configured to run in a Docker container.

**Note**
Connectors, local device and volume resources, and local machine learning model resources can't be used in a Docker container. These features aren't supported when the Lambda runtime environment for the Greengrass group is set to No container (p. 181), which is required to run AWS IoT Greengrass in a Docker container.
You can download a Dockerfile through Amazon CloudFront (p. 21) that has the AWS IoT Greengrass Core software and dependencies installed. To modify the Docker image to run on different platform architectures or reduce the size of the Docker image, see the README file in the Docker package download.

To help you get started quickly and experiment with AWS IoT Greengrass, AWS also provides a prebuilt Docker image that has the AWS IoT Greengrass Core software and dependencies installed. You can download the prebuilt image from Docker Hub or Amazon Elastic Container Registry (Amazon ECR) and run it on Windows, macOS, and Linux (x86_64) platforms. This topic describes how to download the image from Amazon ECR. It contains the following steps:

1. Get the AWS IoT Greengrass Container Image from Amazon ECR (p. 190)
2. Create and Configure the Greengrass Group and Core (p. 193)
3. Run AWS IoT Greengrass Locally (p. 193)
4. Configure "No container" Containerization for the Group (p. 197)
5. Deploy Lambda Functions to the Docker Container (p. 197)
6. (Optional) Deploy Devices that Interact with Greengrass in the Docker Container (p. 197)

Prerequisites

To complete this tutorial, the following software and versions must be installed on your host computer.

- **Docker**, version 18.09 or later. Earlier versions might also work, but version 18.09 or later is preferred.
- **Python**, version 3.6 or later.
- **pip** version 18.1 or later.
- **AWS CLI** version 1.16 or later.
  - To install and configure the CLI, see Installing the AWS Command Line Interface and Configuring the AWS CLI in the AWS Command Line Interface User Guide.
  - To upgrade to the latest version of the AWS CLI, run the following command:

    ```
    pip install awscli --upgrade --user
    ```

**Note**

If you use the MSI installation of the AWS CLI on Windows, be aware of the following:

- If the installation fails to install botocore, try using the Python and pip installation.
- To upgrade to a newer CLI version, you must repeat the MSI installation process.

Step 1: Get the AWS IoT Greengrass Container Image from Amazon ECR

AWS IoT Greengrass provides a Docker image that has the AWS IoT Greengrass Core software installed. For steps that show how to pull the container image from Amazon ECR, choose your operating system:

**Pull the Container Image (Linux)**

Run the following commands in your computer terminal.

1. Get the required login command, which contains an authorization token for the AWS IoT Greengrass registry in Amazon ECR.

    ```
    aws ecr get-login --registry-ids 216483018798 --no-include-email --region us-west-2
    ```
The output is the `docker login` command that you use in the next step.

2. Authenticate your Docker client to the AWS IoT Greengrass container image in the registry by running the `docker login` command from the `get-login` output. The command should be similar to the following example.

   ```bash
docker login -u AWS -p abCzYZ123... https://216483018798.dkr.ecr.us-west-2.amazonaws.com
   ```

3. Retrieve the AWS IoT Greengrass container image.

   ```bash
docker pull 216483018798.dkr.ecr.us-west-2.amazonaws.com/aws-iot-greengrass:latest
   ```

   **Note**
   The `latest` tag corresponds to the latest AWS IoT Greengrass container. You can also pull other versions from the repository. To list all images that are available in the AWS IoT Greengrass repository, use the `aws ecr list-images` command. For example:

   ```bash
aws ecr list-images --region us-west-2 --registry-id 216483018798 --repository-name aws-iot-greengrass
   ```

4. Enable symlink and hardlink protection. If you're experimenting with running AWS IoT Greengrass in a container, you can enable the settings for the current boot only.

   **Note**
   You might need to use `sudo` to run these commands.

   - To enable the settings for the current boot only:

     ```bash
     echo 1 > /proc/sys/fs/protected_hardlinks
     echo 1 > /proc/sys/fs/protected_symlinks
     ```

   - To enable the settings to persist across restarts:

     ```bash
     echo '# AWS Greengrass' >> /etc/sysctl.conf
     echo 'fs.protected_hardlinks = 1' >> /etc/sysctl.conf
     echo 'fs.protected_symlinks = 1' >> /etc/sysctl.conf
     sysctl -p
     ```

5. Enable IPv4 network forwarding, which is required for AWS IoT Greengrass cloud deployment and MQTT communications to work on Linux. In the `/etc/sysctl.conf` file, set `net.ipv4.ip_forward` to 1, and then reload sysctls.

   ```bash
sudo nano /etc/sysctl.conf
# set this net.ipv4.ip_forward = 1
sudo sysctl -p
   ```

   **Note**
   You can use the editor of your choice instead of nano.

---

**Pull the Container Image (macOS)**

Run the following commands in your computer terminal.

1. Get the required login command, which contains an authorization token for the AWS IoT Greengrass registry in Amazon ECR.
Get the AWS IoT Greengrass Container Image from Amazon ECR

aws ecr get-login --registry-ids 216483018798 --no-include-email --region us-west-2

The output is the `docker login` command that you use in the next step.

2. Authenticate your Docker client to the AWS IoT Greengrass container image in the registry by running the `docker login` command from the `get-login` output. The command should be similar to the following example.

```
docker login -u AWS -p abCzYZ123... https://216483018798.dkr.ecr.us-west-2.amazonaws.com
```

3. Retrieve the AWS IoT Greengrass container image.

```
docker pull 216483018798.dkr.ecr.us-west-2.amazonaws.com/aws-iot-greengrass:latest
```

**Note**
The latest tag corresponds to the latest AWS IoT Greengrass container. You can also pull other versions from the repository. To list all images that are available in the AWS IoT Greengrass repository, use the `aws ecr list-images` command. For example:

```
aws ecr list-images --region us-west-2 --registry-id 216483018798 --repository-name aws-iot-greengrass
```

## Pull the Container Image (Windows)

Run the following commands in a command prompt. Before you can use Docker commands on Windows, Docker Desktop must be running.

1. Get the required login command, which contains an authorization token for the AWS IoT Greengrass registry in Amazon ECR.

```
aws ecr get-login --registry-ids 216483018798 --no-include-email --region us-west-2
```

The output is the `docker login` command that you use in the next step.

2. Authenticate your Docker client to the AWS IoT Greengrass container image in the registry by running the `docker login` command from the `get-login` output. The command should be similar to the following example.

```
docker login -u AWS -p abCzYZ123... https://216483018798.dkr.ecr.us-west-2.amazonaws.com
```

3. Retrieve the AWS IoT Greengrass container image.

```
docker pull 216483018798.dkr.ecr.us-west-2.amazonaws.com/aws-iot-greengrass:latest
```

**Note**
The latest tag corresponds to the latest AWS IoT Greengrass container. You can also pull other versions from the repository. To list all images that are available in the AWS IoT Greengrass repository, use the `aws ecr list-images` command. For example:

```
aws ecr list-images --region us-west-2 --registry-id 216483018798 --repository-name aws-iot-greengrass
```
Step 2: Create and Configure the Greengrass Group and Core

The Docker image has the AWS IoT Greengrass Core software installed, but you must create a Greengrass group and core. This includes downloading certificates and the core configuration file.

- Follow the steps in the section called "Configure AWS IoT Greengrass on AWS IoT" (p. 77). Skip the step where you download the AWS IoT Greengrass Core software. The software and its runtime dependencies are already set up in the Docker image.

Step 3: Run AWS IoT Greengrass Locally

After your group is configured, you’re ready to configure and start the core. For steps that show how to do this, choose your operating system:

Run Greengrass Locally (Linux)

Run the following commands in your computer terminal.

1. Decompress the certificates and configuration file (that you downloaded when you created your Greengrass group) into a known location, such as /tmp. For example:

   ```
   tar xzvf hash-setup.tar.gz -C /tmp/
   ```

2. Review Server Authentication in the AWS IoT Developer Guide and choose the appropriate root CA certificate. We recommend that you use Amazon Trust Services (ATS) endpoints and ATS root CA certificates.

   Run the following commands to download the root CA certificate to the directory where you decompressed the certificates and configuration file. Certificates enable your device to connect to AWS IoT over TLS.

   Replace /tmp with the path to the directory.

   **Important**
   
   Your root CA certificate type must match your endpoint. Use an ATS root CA certificate with an ATS endpoint (preferred) or a Verisign root CA certificate with a legacy endpoint. Only some AWS Regions support legacy endpoints. For more information, see the section called "Endpoints Must Match the Certificate Type" (p. 45).

   - For ATS endpoints (preferred), download the appropriate ATS root CA certificate. The following example downloads AmazonRootCA1.pem.

     ```
     cd /tmp/certs/
     sudo wget -O root.ca.pem https://www.amazontrust.com/repository/AmazonRootCA1.pem
     ```

   - For legacy endpoints, download a Verisign root CA certificate. Although legacy endpoints are acceptable for the purposes of this tutorial, we recommend that you create an ATS endpoint and download an ATS root CA certificate.

     ```
     cd /tmp/certs/
     ```
3. Start AWS IoT Greengrass and bind-mount the certificates and configuration file in the Docker container.

   Replace /tmp with the path where you decompressed your certificates and configuration file.

   ```
docker run --rm --init --it --name aws-iot-greengrass \
   --entrypoint /greengrass-entrypoint.sh \
   -v /tmp/certs:/greengrass/certs \
   -v /tmp/config:/greengrass/config \
   -p 8883:8883 \
   216483018798.dkr.ecr.us-west-2.amazonaws.com/aws-iot-greengrass:latest
   
   The output should look like this example:

   Setting up greengrass daemon
   Validating hardlink/softlink protection
   Waiting for up to 30s for Daemon to start

   Greengrass successfully started with PID: 10
   ```

### Run Greengrass Locally (macOS)

Run the following commands in your computer terminal.

1. Decompress the certificates and configuration file (that you downloaded when you created your Greengrass group) into a known location, such as /tmp. For example:

   ```
tar xzvf hash-setup.tar.gz -C /tmp/
   ```

2. Review Server Authentication in the *AWS IoT Developer Guide* and choose the appropriate root CA certificate. We recommend that you use Amazon Trust Services (ATS) endpoints and ATS root CA certificates.

   Run the following commands to download the root CA certificate to the directory where you decompressed the certificates and configuration file. Certificates enable your device to connect to AWS IoT over TLS.

   Replace /tmp with the path to the directory.

   **Important**
   Your root CA certificate type must match your endpoint. Use an ATS root CA certificate with an ATS endpoint (preferred) or a Verisign root CA certificate with a legacy endpoint. Only some AWS Regions support legacy endpoints. For more information, see the section called “Endpoints Must Match the Certificate Type” (p. 45).

   • For ATS endpoints (preferred), download the appropriate ATS root CA certificate. The following example downloads AmazonRootCA1.pem.

   ```
cd /tmp/certs/
   sudo wget -O root.ca.pem https://www.amazontrust.com/repository/AmazonRootCA1.pem
   ```
• For legacy endpoints, download a Verisign root CA certificate. Although legacy endpoints are acceptable for the purposes of this tutorial, we recommend that you create an ATS endpoint and download an ATS root CA certificate.

```bash
cd /tmp/certs/
```

**Note**
The `wget -O` parameter is the capital letter O.

3. Start AWS IoT Greengrass and bind-mount the certificates and configuration file in the Docker container.

Replace `/tmp` with the path where you decompressed your certificates and configuration file.

```bash
```

The output should look like this example:

```
Setting up greengrass daemon
Validating hardlink/softlink protection
Waiting for up to 30s for Daemon to start
Greengrass successfully started with PID: 10
```

**Run Greengrass Locally (Windows)**

1. Use a utility such as WinZip or 7-Zip to decompress the certificates and configuration file that you downloaded when you created your Greengrass group. For more information, see the WinZip documentation.

Locate the downloaded `hash-setup.tar.gz` file on your computer and then decompress the file into `C:\Users\%USERNAME%\Downloads\`

2. Review Server Authentication in the *AWS IoT Developer Guide* and choose the appropriate root CA certificate. We recommend that you use Amazon Trust Services (ATS) endpoints and ATS root CA certificates.

Run the following commands to download the root CA certificate to the directory where you decompressed the certificates and configuration file. Certificates enable your device to connect to AWS IoT over TLS.

**Important**
Your root CA certificate type must match your endpoint. Use an ATS root CA certificate with an ATS endpoint (preferred) or a Verisign root CA certificate with a legacy endpoint. Only some AWS Regions support legacy endpoints. For more information, see the section called "Endpoints Must Match the Certificate Type" (p. 45).

- For ATS endpoints (preferred), download the appropriate ATS root CA certificate. The following example downloads `AmazonRootCA1.pem`.
- If you have `curl` installed, run the following commands in your command prompt.
Run AWS IoT Greengrass Locally

```
cd C:\Users\%USERNAME%\Downloads\certs
curl https://www.amazontrust.com/repository/AmazonRootCA1.pem -o root.ca.pem
```

- Otherwise, in a web browser, open the Amazon Root CA 1 certificate. Save the document as `root.ca.pem` in the `C:\Users\%USERNAME%\Downloads\certs` directory, which contains the decompressed certificates.

  **Note**
  Depending on your browser, save the file directly from the browser or copy the displayed key to the clipboard and save it in Notepad.

- For legacy endpoints, download a Verisign root CA certificate. Although legacy endpoints are acceptable for the purposes of this tutorial, we recommend that you create an ATS endpoint and download an ATS root CA certificate.

- If you have `curl` installed, run the following commands in your command prompt.

```
cd C:\Users\%USERNAME%\Downloads\certs
curl https://www.symantec.com/content/en/us/enterprise/verisign/roots/VeriSign-Class%203-Public-Primary-Certification-Authority-G5.pem -o root.ca.pem
```

  - Otherwise, in a web browser, open the Verisign Class 3 Public Primary G5 root CA certificate. Save the document as `root.ca.pem` in the `C:\Users\%USERNAME%\Downloads\certs` directory, which contains the decompressed certificates.

    **Note**
    Depending on your browser, save the file directly from the browser or copy the displayed key to the clipboard and save it in Notepad.

3. Start AWS IoT Greengrass and bind-mount the certificates and configuration file in the Docker container. Run the following commands in your command prompt.

```
```

When Docker prompts you to share your `C:\` drive with the Docker daemon, allow it to bind-mount the `C:\` directory inside the Docker container. For more information, see Shared drives in the Docker documentation.

The output should look like this example:

```
Setting up greengrass daemon
Validating hardlink/softlink protection
Waiting for up to 30s for Daemon to start
Greengrass successfully started with PID: 10
```

**Note**
If the container doesn't open the shell and exits immediately, you can debug the issue by bind-mounting the Greengrass runtime logs when you start the image. For more information, see the section called “To Persist Greengrass Runtime Logs Outside of the Docker Container” (p. 198).
Step 4: Configure "No container" Containerization for the Greengrass Group

When you run AWS IoT Greengrass in a Docker container, all Lambda functions must run without containerization. In this step, you set the default containerization for the group to No container. You must do this before you deploy the group for the first time.

1. In the AWS IoT console, choose Greengrass, and then choose Groups.
2. Choose the group whose settings you want to change.
3. Choose Settings.
4. Under Lambda runtime environment, choose No container.
5. Choose Update default Lambda execution configuration. Review the message in the confirmation window, and then choose Continue.

For more information, see the section called “Setting Default Containerization for Lambda Functions in a Group” (p. 184).

Note
By default, Lambda functions use the group containerization setting. If you override the No container setting for any Lambda functions when AWS IoT Greengrass is running in a Docker container, the deployment fails.

Step 5: Deploy Lambda Functions to the AWS IoT Greengrass Docker Container

You can deploy long-lived Lambda functions to the Greengrass Docker container.

- Follow the steps in the section called “Module 3 (Part 1): Lambda Functions on AWS IoT Greengrass” (p. 84) to deploy a long-lived Hello World Lambda function to the container.

Step 6: (Optional) Deploy Devices that Interact with Greengrass Running in the Docker Container

You can also deploy Greengrass devices that interact with AWS IoT Greengrass when it's running in a Docker container.

- Follow the steps in the section called “Module 4: Interacting with Devices in an AWS IoT Greengrass Group” (p. 109) to deploy devices that connect to the core and send MQTT messages.

Stopping the AWS IoT Greengrass Docker Container

To stop the AWS IoT Greengrass Docker container, press Ctrl+C in your terminal or command prompt. This action sends SIGTERM to the Greengrass daemon process to tear down the Greengrass daemon process and all Lambda processes that were started by the daemon process. The Docker container is initialized with /dev/init process as PID 1, which helps in removing any leftover zombie processes. For more information, see the Docker run reference.
Troubleshooting AWS IoT Greengrass in a Docker Container

Use the following information to help troubleshoot issues with running AWS IoT Greengrass in a Docker container.

Error: Unknown options: -no-include-email

**Solution:** 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`). If you’re using Windows and you installed the CLI using the MSI installer, you must repeat the installation process. For more information, see Installing the AWS Command Line Interface on Microsoft Windows in the AWS Command Line Interface User Guide.

Warning: IPv4 is disabled. Networking will not work.

**Solution:** You might receive this warning or a similar message when running AWS IoT Greengrass on a Linux computer. Enable IPv4 network forwarding as described in this step (p. 191). AWS IoT Greengrass cloud deployment and MQTT communications don’t work when IPv4 forwarding isn’t enabled. For more information, see Configure namespaced kernel parameters (sysctls) at runtime in the Docker documentation.

Error: A firewall is blocking file Sharing between windows and the containers.

**Solution:** You might receive this error or a Firewall Detected message when running Docker on a Windows computer. See the Error: A firewall is blocking file sharing between Windows and the containers Docker support issue. 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.

For general AWS IoT Greengrass troubleshooting help, see Troubleshooting (p. 516).

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.

To Persist Greengrass Runtime Logs Outside of the Docker Container

You can run the AWS IoT Greengrass Docker container after bind-mounting the `/greengrass/ggc/var/log` directory. The logs persist even after the container exits or is removed.

**On Linux or macOS**

Stop any Greengrass Docker containers (p. 197) running on the host, and then run the following command in a terminal. This bind-mounts the Greengrass log directory and starts the Docker image.

```
docker run --rm --init -it --name aws-iot-greengrass \
    --entrypoint /greengrass-entrypoint.sh \
    -v /tmp/certs:/greengrass/certs \
    -v /tmp/config:/greengrass/config \
```

Replace `/tmp` with the path where you decompressed your certificates and configuration file.
Troubleshooting AWS IoT Greengrass in a Docker Container

You can then check your logs at `/tmp/log` on your host to see what happened while Greengrass was running inside the Docker container.

**On Windows**

Stop any Greengrass Docker containers (p. 197) running on the host, and then run the following command in a command prompt. This bind-mounts the Greengrass log directory and starts the Docker image.

```
cd C:\Users\%USERNAME%\Downloads
mkdir log
```

You can then check your logs at `C:/Users/%USERNAME%/Downloads/log` on your host to see what happened while Greengrass was running inside the Docker container.

**To Attach an Interactive Shell to the Docker Container**

You can attach an interactive shell to a running AWS IoT Greengrass Docker container. This can help you investigate the state of the Greengrass Docker container.

**On Linux or macOS**

While the Greengrass Docker container is running, run the following command in a separate terminal.

```
docker exec -it $(docker ps -a -q -f "name=aws-iot-greengrass") /bin/bash
```

**On Windows**

While the Greengrass Docker container is running, run the following commands in a separate command prompt.

```
docker ps -a -q -f "name=aws-iot-greengrass"
```

Replace `gg-container-id` with the `container_id` result from the previous command.

```
docker exec -it gg-container-id /bin/bash
```
Access Local Resources with Lambda Functions and Connectors

This feature is available for AWS IoT Greengrass Core v1.3 and later.

With AWS IoT Greengrass, you can author AWS Lambda functions and configure connectors (p. 283) in the cloud and deploy them to core devices for local execution. On Greengrass cores running Linux, these locally deployed Lambda functions and connectors can access local resources that are physically present on the Greengrass core device. For example, to communicate with devices that are connected through Modbus or CANbus, you can enable your Lambda function to access the serial port on the core device. To configure secure access to local resources, you must guarantee the security of your physical hardware and your Greengrass core device OS.

To get started accessing local resources, see the following tutorials:

• How to Configure Local Resource Access Using the AWS Command Line Interface (p. 202)
• How to Configure Local Resource Access Using the AWS Management Console (p. 207)

Supported Resource Types

You can access two types of local resources: volume resources and device resources.

Volume resources

Files or directories on the root file system (except under /sys, /dev, or /var). These include:

• Folders or files used to read or write information across Greengrass Lambda functions (for example, /usr/lib/python2.x/site-packages/local).
• Folders or files under the host’s /proc file system (for example, /proc/net or /proc/stat). Supported in v1.6 or later. For additional requirements, see the section called “Volume Resources Under the /proc Directory” (p. 201).

Tip

To configure the /var, /var/run, and /var/lib directories as volume resources, first mount the directory in a different folder and then configure the folder as a volume resource.

When you configure volume resources, you specify a source path and a destination path. The source path is the absolute path of the resource on the host. The destination path is the absolute path of the resource inside the Lambda namespace environment. This is the container that a Greengrass Lambda function or connector runs in. Any changes to the destination path are reflected in the source path on the host file system.

Note

Files in the destination path are visible in the Lambda namespace only. You can’t see them in a regular Linux namespace.

Device resources

Files under /dev. Only character devices or block devices under /dev are allowed for device resources. These include:

• Serial ports used to communicate with devices connected through serial ports (for example, /dev/ttyS0, /dev/ttyS1).
• USB used to connect USB peripherals (for example, /dev/ttyUSB0 or /dev/bus/usb).
• GPIOs used for sensors and actuators through GPIO (for example, /dev/gpio).
Requirements

The following requirements apply to configuring secure access to local resources:

- You must be using AWS IoT Greengrass Core Software v1.3 or later. To create resources for the host's /proc directory, you must be using v1.6 or later.
- The local resource (including any required drivers and libraries) must be correctly installed on the Greengrass core device and consistently available during use.
- The desired operation of the resource, and access to the resource, must not require root privileges.
- Only read or read and write permissions are available. Lambda functions cannot perform privileged operations on the resources.
- You must provide the full path of the local resource on the operating system of the Greengrass core device.
- A resource name or ID has a maximum length of 128 characters and must use the pattern `[a-zA-Z0-9:_-]+`.

Volume Resources Under the /proc Directory

The following considerations apply to volume resources that are under the host's /proc directory.

- You must be using AWS IoT Greengrass Core Software v1.6 or later.
- You can allow read-only access for Lambda functions, but not read-write access. This level of access is managed by AWS IoT Greengrass.
- You might also need to grant OS group permissions to enable read access in the file system. For example, suppose your source directory or file has a 660 file permission, which means that only the owner or user in the group has read (and write) access. In this case, you must add the OS group owner's permissions to the resource. For more information, see the section called “Group Owner File Access Permission” (p. 201).
- The host environment and the Lambda namespace both contain a /proc directory, so be sure to avoid naming conflicts when you specify the destination path. For example, if /proc is the source path, you can specify /host-proc as the destination path (or any path name other than "/proc").

Group Owner File Access Permission

An AWS IoT Greengrass Lambda function process normally runs as ggc_user and ggc_group. However, you can give additional file access permissions to the Lambda function process in the local resource definition, as follows:

- To add the permissions of the Linux group that owns the resource, use the `GroupOwnerSetting#AutoAddGroupOwner` parameter or Automatically add OS group permissions of the Linux group that owns the resource console option.
To add the permissions of a different Linux group, use the `GroupOwnerSetting#GroupOwner` parameter or Specify another OS group to add permission console option. The `GroupOwner` value is ignored if `GroupOwnerSetting#AutoAddGroupOwner` is true.

An AWS IoT Greengrass Lambda function process inherits all of the file system permissions of `ggc_user`, `ggc_group`, and the Linux group (if added). For the Lambda function to access a resource, the Lambda function process must have the required permissions to the resource. You can use the `chmod(1)` command to change the permission of the resource, if necessary.

**See Also**

- AWS IoT Greengrass Limits in the *AWS General Reference*

---

## How to Configure Local Resource Access Using the AWS Command Line Interface

This feature is available for AWS IoT Greengrass Core v1.3 and later.

To use a local resource, you must add a resource definition to the group definition that is deployed to your Greengrass core device. The group definition must also contain a Lambda function definition in which you grant access permissions for local resources to your Lambda functions. For more information, including requirements and constraints, see Access Local Resources with Lambda Functions and Connectors (p. 200).

This tutorial describes the process for creating a local resource and configuring access to it using the AWS Command Line Interface (CLI). To follow the steps in the tutorial, you must have already created a Greengrass group as described in Getting Started with AWS IoT Greengrass (p. 64).

For a tutorial that uses the AWS Management Console, see How to Configure Local Resource Access Using the AWS Management Console (p. 207).

### Create Local Resources

First, you use the `CreateResourceDefinition` command to create a resource definition that specifies the resources to be accessed. In this example, we create two resources, `TestDirectory` and `TestCamera`:

```bash
aws greengrass create-resource-definition --cli-input-json '{
  "Name": "MyLocalVolumeResource",
  "InitialVersion": {
    "Resources": [
      {
        "Id": "data-volume",
        "Name": "TestDirectory",
        "ResourceDataContainer": {
          "LocalVolumeResourceData": {
            "SourcePath": "/src/LRAtest",
            "DestinationPath": "/dest/LRAtest",
            "GroupOwnerSetting": {
              "AutoAddGroupOwner": true,
              "GroupOwner": ""
            }
          }
        }
      }
    ]
  }
}
```
Create Local Resources

The following example shows how to create a local device resource:

```json
{
    "Id": "data-device",
    "Name": "TestCamera",
    "ResourceDataContainer": {
        "LocalDeviceResourceData": {
            "SourcePath": "/dev/video0",
            "GroupOwnerSetting": {
                "AutoAddGroupOwner": true,
                "GroupOwner": ""
            }
        }
    }
}
```

**Resources**: A list of `Resource` objects in the Greengrass group. One Greengrass group can have up to 50 resources.

**Resource#Id**: The unique identifier of the resource. The ID is used to refer to a resource in the Lambda function configuration. Max length 128 characters. Pattern: `[a-zA-Z0-9-._]+`.

**Resource#Name**: The name of the resource. The resource name is displayed in the Greengrass console. Max length 128 characters. Pattern: `[a-zA-Z0-9-._]+`.

**LocalDeviceResourceData#SourcePath**: The local absolute path of the device resource. The source path for a device resource can refer only to a character device or block device under `/dev`.

**LocalVolumeResourceData#SourcePath**: The local absolute path of the volume resource on the Greengrass core device. This location is outside of the container (p. 178) that the function runs in. The source path for a volume resource type cannot start with `/sys`.

**LocalVolumeResourceData#DestinationPath**: The absolute path of the volume resource inside the Lambda environment. This location is inside the container that the function runs in.

**GroupOwnerSetting**: Allows you to configure additional group privileges for the Lambda process. This field is optional. For more information, see Group Owner File Access Permission (p. 201).

**GroupOwnerSetting#AutoAddGroupOwner**: If true, Greengrass automatically adds the specified Linux OS group owner of the resource to the Lambda process privileges. Thus the Lambda process has the file access permissions of the added Linux group.

**GroupOwnerSetting#GroupOwner**: Specifies the name of the Linux OS group whose privileges are added to the Lambda process. This field is optional.

A resource definition version ARN is returned by `CreateResourceDefinition`. The ARN should be used when updating a group definition. For example:

```json
{
    "Name": "MyLocalVolumeResource",
    "LastUpdatedTimestamp": "2017-11-15T01:18:42.153Z",
    "LatestVersion": "a4d9b882-d025-4760-9cfe-9d4fada5390d",
    "CreationTimestamp": "2017-11-15T01:18:42.153Z",
    "Id": "ab14d0b5-116e-4951-a322-9cde24a30373",
    "Arn": "arn:aws:greengrass:us-west-2:123456789012:/greengrass/definition/resources/ab14d0b5-116e-4951-a322-9cde24a30373"
}
Create the Greengrass Function

After the resources are created, use the `CreateFunctionDefinition` command to create the Greengrass function and grant the function access to the resource:

```
aws greengrass create-function-definition --cli-input-json '{
  "Name": "MyFunctionDefinition",
  "InitialVersion": {
    "Functions": [
      {
        "Id": "greengrassLraTest",
        "FunctionConfiguration": {
          "Pinned": false,
          "MemorySize": 16384,
          "Timeout": 30,
          "Environment": {
            "ResourceAccessPolicies": [
              {
                "ResourceId": "data-volume",
                "Permission": "rw"
              },
              {
                "ResourceId": "data-device",
                "Permission": "ro"
              }
            ],
            "AccessSysfs": true
          }
        }
      }
    ]
  }
}
```

**ResourceAccessPolicies**: Contains the `resourceId` and `permission` which grant the Lambda access to the resource. A Lambda function can have a maximum of 10 resources.

**ResourceAccessPolicy#Permission**: Specifies which permissions the Lambda has on the resource. The available options are `rw` (read/write) or `ro` (read-only).

**AccessSysfs**: If true, the Lambda process can have read access to the `/sys` folder on the Greengrass core device. This is used in cases where the Greengrass Lambda needs to read device information from `/sys`.

Again, `CreateFunctionDefinition` returns a function definition version ARN. The ARN should be used in your group definition version.

```
{
  "LatestVersionArn": "arn:aws:greengrass:us-west-2:012345678901:/greengrass/definition/functions/3c9b1685-634f-4592-8f6d-7ae1183c28ad/versions/37f0d50e-ef50-4f5f-b125-ade8ed12336e",
  "Name": "MyFunctionDefinition",
  "LastUpdatedTimestamp": "2017-11-22T02:28:02.325Z",
  "LatestVersion": "37f0d50e-ef50-4f5f-b125-ade8ed12336e",
  "CreationTimestamp": "2017-11-22T02:28:02.325Z",
  "Id": "3c9b1685-634f-4592-8f6d-7ae1183c28ad",
  "Arn": "arn:aws:greengrass:us-west-2:123456789012:/greengrass/definition/functions/3c9b1685-634f-4592-8f6d-7ae1183c28ad"
}
```
Add the Lambda Function to the Group

Finally, use `CreateGroupVersion` to add the function to the group. For example:

```bash
aws greengrass create-group-version --group-id "b36a3aeb-3243-47ff-9fa4-7e8d98cd3cf5" \
greengrass/definition/resources/db6bf40b-29d3-4c4e-9574-21ab7d74316c/versions/31d0010f-
e19a-404c-8098-68b79906fb87" \
greengrass/definition/cores/adbff3475-f6f3-48e1-84d6-502f02729067/
versions/297c419a-9deb-46dd-8ccc-341fc670138b" \
definition/functions/d1123830-da38-4c4c-a4b7-e92ee7b6d3e/versions/a2e90400-caee-f4fd-b23a-
db1892a33c78" \
greengrass/definition/subscriptions/7a8ef3d8-1de3-426c-9554-5b5a32fbc6b6/
versions/470c858c-7eb3-4abd-9d48-230b236bbf6a"
```

A new group version is returned:

```json
{
    "Arn": "arn:aws:greengrass:us-west-2:012345678901:/greengrass/groups/
b36a3aeb-3243-47ff-9fa4-7e8d98cd3cf5/versions/291917fb-ec54-4895-823e-27b52da25481",
    "Version": "291917fb-ec54-4895-823e-27b52da25481",
    "CreationTimestamp": "2017-11-22T01:47:22.487Z",
    "Id": "b36a3aeb-3243-47ff-9fa4-7e8d98cd3cf5"
}
```

Your Greengrass group now contains the `lraTest` Lambda function that has access to two resources: `TestDirectory` and `TestCamera`.

This example Lambda function, `lraTest.py`, written in Python, writes to the local volume resource:

```python
# Demonstrates a simple use case of local resource access.
# This Lambda function writes a file "test" to a volume mounted inside
# the Lambda environment under "/dest/LRAtest". Then it reads the file and
# publishes the content to the AWS IoT "LRA/test" topic.
import sys
import greengrasssdk
import platform
import os
import logging

# Create a Greengrass Core SDK client.
client = greengrasssdk.client('iot-data')
volumePath = '/dest/LRAtest/

def function_handler(event, context):
    try:
        volumeInfo = os.stat(volumePath)
        client.publish(topic='LRA/test', payload=str(volumeInfo))
        with open(volumePath + '/test', 'a') as output:
            output.write('Successfully write to a file.\n')
        with open(volumePath + '/test', 'r') as myfile:
            data = myfile.read()
        client.publish(topic='LRA/test', payload=data)
    except Exception as err:
        logging.error(err)

client = greengrasssdk.client('iot-data')
volumePath = '/dest/LRAtest/
def function_handler(event, context):
    try:
        volumeInfo = os.stat(volumePath)
        client.publish(topic='LRA/test', payload=str(volumeInfo))
        with open(volumePath + '/test', 'a') as output:
            output.write('Successfully write to a file.\n')
        with open(volumePath + '/test', 'r') as myfile:
            data = myfile.read()
        client.publish(topic='LRA/test', payload=data)
    except Exception as err:
        logging.error(err)
```
except Exception as e:
    logging.error("Experiencing error :\{0\}".format(e))
return

These commands are provided by the Greengrass API to create and manage resource definitions and resource definition versions:

- CreateResourceDefinition
- CreateResourceDefinitionVersion
- DeleteResourceDefinition
- GetResourceDefinition
- GetResourceDefinitionVersion
- ListResourceDefinitions
- ListResourceDefinitionVersions
- UpdateResourceDefinition

**Troubleshooting**

- **Q:** Why does my Greengrass group deployment fail with an error similar to:

  ```
  group config is invalid:
  gcc_user or [ggc_group root tty] don't have ro permission on the file: /dev/tty0
  ```

  **A:** This error indicates that the Lambda process doesn't have permission to access the specified resource. The solution is to change the file permission of the resource so that Lambda can access it. (See Group Owner File Access Permission (p. 201) for details).

- **Q:** When I configure `/var/run` as a volume resource, why does the Lambda function fail to start with an error message in the runtime.log:

  ```
  [ERROR]-container_process.go:39, Runtime execution error: unable to start lambda container.
  container_linux.go:259: starting container process caused "process_linux.go:345: container init caused "rootfs_linux.go:62: mounting \"/var/run\" to rootfs \"/\" at \"/greengrass/ggc/packages/1.3.0/rootfs_sys\" caused \"invalid argument\""
  ```

  **A:** AWS IoT Greengrass core currently doesn't support the configuration of `/var`, `/var/run`, and `/var/lib` as volume resources. One workaround is to first mount `/var`, `/var/run` or `/var/lib` in a different folder and then configure the folder as a volume resource.

- **Q:** When I configure `/dev/shm` as a volume resource with read-only permission, why does the Lambda function fail to start with an error in the runtime.log:

  ```
  [ERROR]-container_process.go:39, Runtime execution error: unable to start lambda container.
  container_linux.go:259: starting container process caused "process_linux.go:345: container init caused "rootfs_linux.go:62: mounting \"/dev/shm\" to rootfs \"/\" at \"/greengrass/ggc/packages/1.3.0/rootfs_sys\" caused \"operation not permitted\""
  ```

  **A:** `/dev/shm` can only be configured as read/write. Change the resource permission to `rw` to resolve the issue.
This feature is available for AWS IoT Greengrass Core v1.3 and later.

You can configure Lambda functions to securely access local resources on the host Greengrass core device. Local resources refer to buses and peripherals that are physically present on the host, or file system volumes on the host OS. For more information, including requirements and constraints, see Access Local Resources with Lambda Functions and Connectors (p. 200).

This tutorial describes how to use the AWS Management Console to configure access to local resources that are present on an AWS IoT Greengrass core device. It contains the following high-level steps:

1. Create a Lambda Function Deployment Package (p. 207)
2. Create and Publish a Lambda Function (p. 208)
3. Add the Lambda Function to the Group (p. 211)
4. Add a Local Resource to the Group (p. 213)
5. Add Subscriptions to the Group (p. 214)
6. Deploy the Group (p. 216)

For a tutorial that uses the AWS Command Line Interface, see How to Configure Local Resource Access Using the AWS Command Line Interface (p. 202).

**Prerequisites**

To complete this tutorial, you need:

- A Greengrass group and a Greengrass core (v1.3 or later). To create a Greengrass group or core, see Getting Started with AWS IoT Greengrass (p. 64).
- The following directories on the Greengrass core device:
  - `/src/LRAtest`
  - `/dest/LRAtest`

The owner group of these directories must have read and write access to the directories. You might use the following command to grant access:

```
sudo chmod 0775 /src/LRAtest
```

**Step 1: Create a Lambda Function Deployment Package**

In this step, you create a Lambda function deployment package, which is a ZIP file that contains the function's code and dependencies. You also download the AWS IoT Greengrass Core SDK to include in the package as a dependency.

1. On your computer, copy the following Python script to a local file named `lraTest.py`. This is the app logic for the Lambda function.

```
# Demonstrates a simple use case of local resource access.
```
import sys
import greengrasssdk
import platform
import os
import logging

# Create a Greengrass Core SDK client.
client = greengrasssdk.client('iot-data')
volumePath = '/dest/LRA/test'

def function_handler(event, context):
    client.publish(topic='LRA/test', payload='Sent from AWS IoT Greengrass Core.')
    try:
        volumeInfo = os.stat(volumePath)
        client.publish(topic='LRA/test', payload=str(volumeInfo))
        with open(volumePath + '/test', 'a') as output:
            output.write('Successfully write to a file.
')
        with open(volumePath + '/test', 'r') as myfile:
            data = myfile.read()
        client.publish(topic='LRA/test', payload=data)
    except Exception as e:
        logging.error("Experiencing error :{}").format(e)
    return

2. Download the AWS IoT Greengrass Core SDK for Python from the AWS IoT Greengrass Core SDK (p. 22) downloads page.
3. Unzip the downloaded package to get the SDK. The SDK is the greengrasssdk folder.
4. Zip the following items into a file named lraTestLambda.zip:
   - lraTest.py. App logic.
   - greengrasssdk. Required library for all Python Lambda functions.

The lraTestLambda.zip file is your Lambda function deployment package. Now you’re ready to create a Lambda function and upload the deployment package.

Step 2: Create and Publish a Lambda Function

In this step, you use the AWS Lambda console to create a Lambda function and configure it to use your deployment package. Then, you publish a function version and create an alias.

First, create the Lambda function.

1. In the AWS Management Console, choose Services, and open the AWS Lambda console.
2. Choose Create function.
3. Choose Author from scratch.
4. In the Basic information section, use the following values.
   a. For Function name, enter TestLRA.
   b. For Runtime, choose Python 2.7.
   c. For Permissions, keep the default setting. This creates an execution role that grants basic Lambda permissions. This role isn’t used by AWS IoT Greengrass.
5. Choose Create function.
Now, upload your Lambda function deployment package and register the handler.

6. On the **Configuration** tab for the function, in **Function code**, use the following values.

   a. For **Code entry type**, choose **Upload a .zip file**.
   b. For **Runtime**, choose **Python 2.7**.
   c. For **Handler**, enter **lraTest.function_handler**.

7. Choose **Upload**.

8. Choose your **lraTestLambda.zip** deployment package.

9. At the top of the page, choose **Save**.

   ![Lambda Function Configuration](image)

**Note**

The **Test** button on the AWS Lambda console doesn't work with this function. The AWS IoT Greengrass Core SDK doesn't contain modules that are required to run your Greengrass Lambda functions independently in the AWS Lambda console. These modules (for example, `greengrass_common`) are supplied to the functions after they are deployed to your Greengrass core.

You can see your code in the **Function code** section by choosing **Edit code inline** from the **Code entry type** menu.

Next, publish the first version of your Lambda function. Then, create an alias for the version.
Greengrass groups can reference a Lambda function by alias (recommended) or by version. Using an alias makes it easier to manage code updates because you don't have to change your subscription table or group definition when the function code is updated. Instead, you just point the alias to the new function version.

10. From Actions, choose Publish new version.

11. For Version description, enter First version, and then choose Publish.

12. On the TestLRA: 1 configuration page, from Actions, choose Create alias.

13. On the Create a new alias page, for Name, enter test. For Version, enter 1.

**Note**
AWS IoT Greengrass doesn't support Lambda aliases for $LATEST versions.

14. Choose Create.

You can now add the Lambda function to your Greengrass group.
Step 3: Add the Lambda Function to the Greengrass Group

In this step, you add the function to your group and configure the function's lifecycle.

First, add the Lambda function to your Greengrass group.

1. In the AWS IoT console, choose Greengrass, and then choose Groups.

2. Choose the Greengrass group where you want to add the Lambda function.

3. On the group configuration page, choose Lambdas, and then choose Add Lambda.

4. On the Add a Lambda to your Greengrass Group page, choose Use existing Lambda.
Add the Lambda Function to the Group

5. On the **Use existing Lambda** page, choose **TestLRA**, and then choose **Next**.
6. On the **Select a Lambda version** page, choose **Alias:test**, and then choose **Finish**.

Next, configure the lifecycle of the Lambda function.

7. On the **Lambdas** page, choose the TestLRA Lambda function.

8. On the **TestLRA** configuration page, choose **Edit**.
9. On the **Group-specific Lambda configuration** page, use the following values.
   a. For **Timeout**, choose **30 seconds**.
   b. Under **Lambda lifecycle**, select **Make this function long-lived and keep it running indefinitely**. For more information, see the section called “Lifecycle Configuration” (p. 187).

**Important**
Lambda functions that use local resources (as described in this procedure) must run in a Greengrass container. Otherwise, deployment fails if you try to deploy the function. For more information, see **Containerization** (p. 178).

10. At the bottom of the page, choose **Update**.
Step 4: Add a Local Resource to the Greengrass Group

In this step, you add a local volume resource to the Greengrass group and grant the function read and write access to the resource. A local resource has a group-level scope. You can grant permissions for any Lambda function in the group to access the resource.

1. On the group configuration page, choose Resources.

   Deployments
   Subscriptions
   Cores
   Devices
   Lambdas
   Resources
   Connectors
   Tags
   Settings

2. On the Local tab, choose Add a local resource.
3. On the Create a local resource page, use the following values.
   a. For Resource name, enter testDirectory.
   b. For Resource type, choose Volume.
   c. For Source path, enter /src/LRAtest. This path must exist on the host OS.
      The source path is the local absolute path of the resource on the file system of the core device. This location is outside of the container (p. 178) that the function runs in. The path can't start with /sys.
   d. For Destination path, enter /dest/LRAtest. This path must exist on the host OS.
      The destination path is the absolute path of the resource in the Lambda namespace. This location is inside the container that the function runs in.
   e. Under Group owner file access permission, select Automatically add OS group permissions of the Linux group that owns the resource.
      The Group owner file access permission option lets you grant additional file access permissions to the Lambda process. For more information, see Group Owner File Access Permission (p. 201).
Step 5: Add Subscriptions to the Greengrass Group

In this step, you add two subscriptions to the Greengrass group. These subscriptions enable bidirectional communication between the Lambda function and AWS IoT.

First, create a subscription for the Lambda function to send messages to AWS IoT.

1. On the group configuration page, choose Subscriptions, and then choose Add Subscription.
2. On the **Select your source and target** page, configure the source and target, as follows:
   
   a. For **Select a source**, choose *Lambdas*, and then choose *TestLRA*.
   
   b. For **Select a target**, choose *Services*, and then choose *IoT Cloud*.
   
   c. Choose **Next**.

3. On the **Filter your data with a topic** page, for **Optional topic filter**, enter *LRA/test*, and then choose **Next**.

4. Choose **Finish**. The **Subscriptions** page displays the new subscription.

   Next, configure a subscription that invokes the function from AWS IoT.

5. On the **Subscriptions** page, choose **Add Subscription**.
6. On the **Select your source and target** page, configure the source and target, as follows:
   a. For **Select a source**, choose **Services**, and then choose **IoT Cloud**.
   b. For **Select a target**, choose **Lambdas**, and then choose **TestLRA**.
   c. Choose **Next**.

![Select a source and target](image1)

7. On the **Filter your data with a topic** page, for **Optional topic filter**, enter **invoke/LRAFunction**, and then choose **Next**.

![Filter your data with a topic](image2)

8. Choose **Finish**. The **Subscriptions** page displays both subscriptions.

---

**Step 6: Deploy the AWS IoT Greengrass Group**

In this step, you deploy the current version of the group definition.

1. Make sure that the AWS IoT Greengrass core is running. Run the following commands in your Raspberry Pi terminal, as needed.
   a. To check whether the daemon is running:

   ```bash
   ps aux | grep -E 'greengrass.*daemon'
   ```

   If the output contains a root entry for `/greengrass/ggc/packages/1.9.4/bin/daemon`, then the daemon is running.
Test Local Resource Access

Now you can verify whether the local resource access is configured correctly. To test, you subscribe to the LRA/test topic and publish to the invoke/LRAFunction topic. The test is successful if the Lambda function sends the expected payload to AWS IoT.
1. On the AWS IoT console home page, in the left pane, choose Test.

2. In the Subscriptions section, for Subscription topic, enter LRA/test. For MQTT payload display, select Display payloads as strings.

3. Choose Subscribe to topic. Your Lambda function publishes to the LRA/test topic.
4. In the **Publish** section, enter `invoke/LRAFunction`, and then choose **Publish to topic** to invoke your Lambda function. The test is successful if the page displays the function’s three message payloads.
The test file created by the Lambda function is in the `/src/LRAtest` directory on the Greengrass core device. Although the Lambda function writes to a file in the `/dest/LRAtest` directory, that file is visible in the Lambda namespace only. You can't see it in a regular Linux namespace. Any changes to the destination path are reflected in the source path on the file system.

For troubleshooting help, see *Troubleshooting (p. 516).*
Perform Machine Learning Inference

This feature is available for AWS IoT Greengrass Core v1.6 or later.

With AWS IoT Greengrass, you can perform machine learning (ML) inference at the edge 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.

To get started performing local inference, see the section called “How to Configure Machine Learning Inference” (p. 226).

How AWS IoT Greengrass ML Inference Works

You can train your inference models anywhere, deploy them locally as machine learning resources in a Greengrass group, and then access them from Greengrass Lambda functions. For example, you can build and train deep-learning models in Amazon SageMaker and deploy them to your Greengrass core. Then, your Lambda functions can use the local models to perform inference on connected devices and send new training data back to the cloud.

The following diagram shows the AWS IoT Greengrass ML inference workflow.

AWS IoT Greengrass ML inference simplifies each step of the ML workflow, including:

- Building and deploying ML framework prototypes.
- Accessing cloud-trained models and deploying them to Greengrass core devices.
- Creating inference apps that can access hardware accelerators (such as GPUs and FPGAs) as local resources (p. 200).

Machine Learning Resources

Machine learning resources represent cloud-trained inference models that are deployed to an AWS IoT Greengrass core. To deploy machine learning resources, first you add the resources to a Greengrass
group, and then you define how Lambda functions in the group can access them. During group deployment, AWS IoT Greengrass retrieves the source model packages from the cloud and extracts them to directories inside the Lambda runtime namespace. Then, Greengrass Lambda functions use the locally deployed models to perform inference.

To update a locally deployed model, first update the source model (in the cloud) that corresponds to the machine learning resource, and then deploy the group. During deployment, AWS IoT Greengrass checks the source for changes. If changes are detected, then AWS IoT Greengrass updates the local model.

**Supported Model Sources**

AWS IoT Greengrass supports Amazon SageMaker and Amazon S3 model sources for machine learning resources.

The following requirements apply to model sources:

- S3 buckets that store your Amazon SageMaker and Amazon S3 model sources must not be encrypted using SSE-C. For buckets that use server-side encryption, AWS IoT Greengrass ML 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 Developer Guide*.

- The names of S3 buckets that store your Amazon SageMaker and Amazon S3 model sources 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 Developer Guide*.

- Service-level AWS Region support must be available for both AWS IoT Greengrass and Amazon SageMaker. See the following table.

<table>
<thead>
<tr>
<th>AWS Region</th>
<th>Amazon SageMaker Models</th>
<th>Amazon S3 Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>US East (Ohio)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>US East (N. Virginia)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>US West (Oregon)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Asia Pacific (Mumbai)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Asia Pacific (Seoul)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Asia Pacific (Singapore)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Asia Pacific (Sydney)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Asia Pacific (Tokyo)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>EU (Frankfurt)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>EU (Ireland)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>EU (London)</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
AWS IoT Greengrass must have read permission to the model source, as described in the following sections.

Amazon SageMaker

AWS IoT Greengrass supports models that are saved as Amazon SageMaker training jobs.

If you configured your Amazon SageMaker environment by creating a bucket whose name contains sagemaker, then AWS IoT Greengrass has sufficient permission to access your Amazon SageMaker training jobs. The AWSGreengrassResourceAccessRolePolicy managed policy allows access to buckets whose name contains the string sagemaker. This policy is attached to the Greengrass service role.

Otherwise, you must grant AWS IoT Greengrass read permission to the bucket where your training job is stored. To do this, embed the following inline policy in the Greengrass service role. You can list multiple bucket ARNs.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": [
                "arn:aws:s3:::my-bucket-name"
            ]
        }
    ]
}
```

Amazon SageMaker is a fully managed ML service that you can use to build and train models using built-in or custom algorithms. For more information, see What Is Amazon SageMaker? in the Amazon SageMaker Developer Guide.

Amazon S3

AWS IoT Greengrass supports models that are stored in Amazon S3 as tar.gz or .zip files.

To enable AWS IoT Greengrass to access models that are stored in Amazon S3 buckets, you must grant AWS IoT Greengrass read permission to access the buckets by doing one of the following:

- Store your model in a bucket whose name contains greengrass.

  The AWSGreengrassResourceAccessRolePolicy managed policy allows access to buckets whose name contains the string greengrass. This policy is attached to the Greengrass service role.

- Embed an inline policy in the Greengrass service role.

  If your bucket name doesn't contain greengrass, add the following inline policy to the service role. You can list multiple bucket ARNs.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": [
                "arn:aws:s3:::my-bucket-name"
            ]
        }
    ]
}
```
Requirements

The following requirements apply for creating and using machine learning resources:

- You must be using AWS IoT Greengrass Core v1.6 or later.
- Lambda functions can perform read or read and write operations on the resource. Permissions for other operations are not available.
- Lambda functions configured with access to ML model resources cannot be running in non-containerized mode.
- You must provide the full path of the resource on the operating system of the core device.
- A resource name or ID has a maximum length of 128 characters and must use the pattern `[a-zA-Z0-9:_-]+`.

Runtimes and Precompiled Framework Libraries for ML Inference

To help you quickly get started experimenting with ML inference, AWS IoT Greengrass provides runtimes and precompiled framework libraries.

- Amazon SageMaker Neo deep learning runtime (p. 224) (Greengrass Core Software License Agreement)
- Apache MXNet (Apache License 2.0)
- TensorFlow (Apache License 2.0)
- Chainer (MIT License)

These runtimes and precompiled libraries can be installed on NVIDIA Jetson TX2, Intel Atom, and Raspberry Pi platforms. The runtimes and libraries are available from the AWS IoT Greengrass Machine Learning Runtimes and Precompiled Libraries (p. 22) downloads page. You can install them directly on your core or include them as part of the software in your Greengrass group.

Be sure to read the following information about compatibility and limitations.

Amazon SageMaker Neo Deep Learning Runtime

You can use the Amazon SageMaker Neo deep learning runtime to perform inference with optimized machine learning models on your AWS IoT Greengrass devices. These models are optimized using the Amazon SageMaker Neo deep learning compiler to improve machine learning inference prediction.
speeds. For more information about model optimization in Amazon SageMaker, see the Amazon SageMaker Neo documentation.

Note
Currently, you can optimize machine learning models using the Neo deep learning compiler in specific AWS Regions only. However, you can use the Neo deep learning runtime with optimized models in all AWS Regions where AWS IoT Greengrass core is supported. For information, see How to Configure Optimized Machine Learning Inference (p. 244).

MXNet Versioning
Apache MXNet doesn’t currently ensure forward compatibility, so models that you train using later versions of the framework might not work properly in earlier versions of the framework. To avoid conflicts between the model-training and model-serving stages, and to provide a consistent end-to-end experience, use the same MXNet framework version in both stages.

MXNet on Raspberry Pi
Greengrass Lambda functions that access local MXNet models must set the following environment variable:

MXNET_ENGINE_TYPE=NaiveEngine

You can set the environment variable in the function code or add it to the function’s group-specific configuration. For an example that adds it as a configuration setting, see this step (p. 232).

Note
For general use of the MXNet framework, such as running a third-party code example, the environment variable must be configured on the Raspberry Pi.

TensorFlow Model-Serving Limitations on Raspberry Pi
Currently, the AWS IoT Greengrass TensorFlow installer supports installation on 32-bit laptop or desktop operating systems only. To build TensorFlow on 64-bit platforms, see Installing TensorFlow in the TensorFlow documentation.

The following recommendations for improving inference results are based on our tests with the 32-bit Arm precompiled libraries on the Raspberry Pi platform. These recommendations are intended for advanced users for reference only, without guarantees of any kind.

- Models that are trained using the Checkpoint format should be “frozen” to the protocol buffer format before serving. For an example, see the TensorFlow-Slim image classification model library.
- Don’t use the TF-Estimator and TF-Slim libraries in either training or inference code. Instead, use the .pb file model-loading pattern that’s shown in the following example.

```python
graph = tf.Graph()
graph_def = tf.GraphDef()
graph_def.ParseFromString(pb_file.read())
with graph.as_default():
    tf.import_graph_def(graph_def)
```

Note
For more information about supported platforms for TensorFlow, see Installing TensorFlow in the TensorFlow documentation.
How to Configure Machine Learning Inference Using the AWS Management Console

To follow the steps in this tutorial, you must be using AWS IoT Greengrass Core v1.6 or later.

You can perform machine learning (ML) inference locally on a Greengrass core device using data from connected devices. For information, including requirements and constraints, see Perform Machine Learning Inference (p. 221).

This tutorial describes how to use the AWS Management Console to configure a Greengrass group to run a Lambda inference app that recognizes images from a camera locally, without sending data to the cloud. The inference app accesses the camera module on a Raspberry Pi and runs inference using the open source SqueezeNet model.

The tutorial contains the following high-level steps:

2. Install the MXNet Framework (p. 227).
3. Create a model package (p. 228).
4. Create and publish a Lambda function (p. 228).
5. Add the Lambda function to the group (p. 232).
6. Add resources to the group (p. 234).
7. Add a subscription to the group (p. 238).
8. Deploy the group (p. 239).

Prerequisites

To complete this tutorial, you need:

- Raspberry Pi 4 Model B, or Raspberry Pi 3 Model B/B+, set up and configured for use with AWS IoT Greengrass. To learn how to set up your Raspberry Pi with AWS IoT Greengrass, see Module 1 and Module 2 of Getting Started with AWS IoT Greengrass (p. 64).

  Note
  The Raspberry Pi might require a 2.5A power supply to run the deep learning frameworks that are typically used for image classification. A power supply with a lower rating might cause the device to reboot.

- 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.

- A Greengrass group and a Greengrass core. To learn how to create a Greengrass group or core, see Getting Started with AWS IoT Greengrass (p. 64).

  Note
  This tutorial uses a Raspberry Pi, but AWS IoT Greengrass supports other platforms, such as Intel Atom and NVIDIA Jetson TX2 (p. 243). The example for Jetson TX2 can use static images instead of images streamed from a camera.

Step 1: Configure the Raspberry Pi

In this step, you install updates to the Rasbian operating system, install the camera module software and Python dependencies, and enable the camera interface. Run the following commands in your Raspberry Pi terminal.

Prerequisites

To complete this tutorial, you need:

- Raspberry Pi 4 Model B, or Raspberry Pi 3 Model B/B+, set up and configured for use with AWS IoT Greengrass. To learn how to set up your Raspberry Pi with AWS IoT Greengrass, see Module 1 and Module 2 of Getting Started with AWS IoT Greengrass (p. 64).

  Note
  The Raspberry Pi might require a 2.5A power supply to run the deep learning frameworks that are typically used for image classification. A power supply with a lower rating might cause the device to reboot.

- 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.

- A Greengrass group and a Greengrass core. To learn how to create a Greengrass group or core, see Getting Started with AWS IoT Greengrass (p. 64).

  Note
  This tutorial uses a Raspberry Pi, but AWS IoT Greengrass supports other platforms, such as Intel Atom and NVIDIA Jetson TX2 (p. 243). The example for Jetson TX2 can use static images instead of images streamed from a camera.

Step 1: Configure the Raspberry Pi

In this step, you install updates to the Rasbian operating system, install the camera module software and Python dependencies, and enable the camera interface. Run the following commands in your Raspberry Pi terminal.
1. Install updates to Raspbian.

```
sudo apt-get update
dsud apt-get dist-upgrade
```

2. Install the picamera interface for the camera module and other Python libraries that are required for this tutorial.

```
sudo apt-get install -y python-dev python-setuptools python-pip python-picamera
```

3. Reboot the Raspberry Pi.

```
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. Use 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 `/home/pi` directory, and displays information about the camera in the Raspberry Pi terminal.

### Step 2: Install the MXNet Framework

In this step, you download precompiled Apache MXNet libraries and install them on your Raspberry Pi.

**Note**
This tutorial uses libraries for the MXNet ML framework, but libraries for TensorFlow are also available. For more information, including limitations, see the section called “Runtimes and Precompiled Framework Libraries for ML Inference” (p. 224).

1. On the [AWS IoT Greengrass Machine Learning Runtimes and Precompiled Libraries](p. 22) downloads page, locate MXNet version 1.2.1 for Raspberry Pi. Choose **Download**.

   **Note**
   By downloading this software you agree to the Apache License 2.0.

2. Transfer the downloaded `ggc-mxnet-v1.2.1-python-raspi.tar.gz` file from your computer to your Raspberry Pi.

   **Note**
   For ways that you can do this on different platforms, see this step (p. 81) in the Getting Started section. For example, you might use the following `scp` command:

   ```
   scp ggc-mxnet-v1.2.1-python-raspi.tar.gz pi@IP-address:/home/pi
   ```

3. In your Raspberry Pi terminal, unpack the transferred file.

   ```
tar -xzf ggc-mxnet-v1.2.1-python-raspi.tar.gz
```

4. Install the MXNet framework.
AWS IoT Greengrass Developer Guide
Create a Model Package

```bash
cd ggc-mxnet-v1.2.1-python-raspi/
./mxnet_installer.sh
```

**Note**
You can continue to the section called “Create a Model Package” (p. 228) while the framework is being installed, but you must wait until the installation is complete before you proceed to the section called “Create and Publish a Lambda Function” (p. 228).

You can optionally run unit tests to verify the installation. To do so, add the `-u` option to the previous command. If successful, each test logs a line in the terminal that ends with `ok`. If all tests are successful, the final log statement contains `OK`. Running unit tests increases the installation time.

The script also creates a Lambda function deployment package named `greengrassObjectClassification.zip` that contains the function code and dependencies. You upload this deployment package later.

5. When the installation is complete, transfer `greengrassObjectClassification.zip` to your computer. Depending on your environment, you can use the `scp` command or a utility such as `WinSCP`.

### Step 3: Create an MXNet Model Package

In this step, you download files for a sample pretrained MXNet model, and then save them as a `.zip` file. AWS IoT Greengrass can use models from Amazon S3, provided that they use the `.tar.gz` or `.zip` format.

1. Download the following files to your computer:
   - `squeezenet_v1.1-0000.params`. A parameter file that describes weights of the connectivity.
   - `squeezenet_v1.1-symbol.json`. A symbol file that describes the neural network structure.
   - `synset.txt`. A synset file that maps recognized class IDs to human-readable class names.

   **Note**
   All MXNet model packages use these three file types, but the contents of TensorFlow model packages vary.

2. Zip the three files, and name the compressed file `squeezenet.zip`. You upload this model package to Amazon S3 in the section called “Add Resources to the Group” (p. 234).

### Step 4: Create and Publish a Lambda Function

In this step, you create a Lambda function and configure it to use the deployment package. Then, you publish a function version and create an alias.

The Lambda function deployment package is named `greengrassObjectClassification.zip`. This is the `.zip` file that was generated during the MXNet framework installation in Step 2: Install the MXNet Framework (p. 227). It contains an inference app that performs common tasks, such as loading models, importing Apache MXNet, and taking actions based on predictions. The app contains the following key components:

- **App logic**:
  - `load_model.py`. Loads MXNet models.
  - `greengrassObjectClassification.py`. Runs predictions on images that are streamed from the camera.
• Dependencies:
  • greengrassdk. The AWS IoT Greengrass Core SDK for Python, used by the function to publish MQTT messages.

  **Note**
The `mxnet` library was installed on the core device during the MXNet framework installation.

First, create the Lambda function.

1. In the AWS IoT console, in the navigation pane, choose **Greengrass**, and then choose **Groups**.

2. Choose the Greengrass group where you want to add the Lambda function.

3. On the group configuration page, choose **Lambdas**, and then choose **Add Lambda**.
4. On the **Add a Lambda to your Greengrass Group** page, choose **Create new Lambda**. This opens the AWS Lambda console.

   **Create a new Lambda function**
   You will be taken to the AWS Lambda Console and can author a new Lambda function.

   **Use an existing Lambda function**
   You will choose from a list of existing Lambda functions.

5. Choose **Author from scratch** and use the following values to create your function:
   - For **Function name**, enter `greengrassObjectClassification`.
   - For **Runtime**, choose **Python 2.7**.

   For **Permissions**, keep the default setting. This creates an execution role that grants basic Lambda permissions. This role isn't used by AWS IoT Greengrass.

6. Choose **Create function**.

   Now, upload your Lambda function deployment package and register the handler.
7. On the **Configuration** tab for the `greengrassObjectClassification` function, for **Function code**, use the following values:
   - For **Code entry type**, choose **Upload a .zip file**.
   - For **Runtime**, choose **Python 2.7**.
   - For **Handler**, enter `greengrassObjectClassification.function_handler`.

8. Choose **Upload**.

   ![Upload](Image)

9. Choose your `greengrassObjectClassification.zip` deployment package.

10. Choose **Save**.

Next, publish the first version of your Lambda function. Then, create an alias for the version.

**Note**
Greengrass groups can reference a Lambda function by alias (recommended) or by version. Using an alias makes it easier to manage code updates because you don't have to change your subscription table or group definition when the function code is updated. Instead, you just point the alias to the new function version.

11. From the **Actions** menu, choose **Publish new version**.

![Publish new version](Image)

12. For **Version description**, enter **First version**, and then choose **Publish**.

13. On the `greengrassObjectClassification: 1` configuration page, from the **Actions** menu, choose **Create alias**.

![Create alias](Image)

14. On the **Create a new alias** page, use the following values:
   - For **Name**, enter `mlTest`.
   - For **Version**, enter `1`.

   **Note**
AWS IoT Greengrass doesn't support Lambda aliases for `$LATEST` versions.

15. Choose **Create**.
Now, add the Lambda function to your Greengrass group.

**Step 5: Add the Lambda Function to the Greengrass Group**

In this step, you add the Lambda function to the group and then configure its lifecycle and environment variables.

First, add the Lambda function to your Greengrass group.

1. In the AWS IoT console, open the group configuration page.
2. Choose **Lambdas**, and then choose **Add Lambda**.
3. On the **Add a Lambda to your Greengrass Group** page, choose **Use existing Lambda**.
4. Choose `greengrassObjectClassification`, and then choose Next.
5. On the Select a Lambda version page, choose `Alias:mlTest`, and then choose Finish.

Next, configure the lifecycle and environment variables of the Lambda function.

6. On the Lambertas page, choose the `greengrassObjectClassification` Lambda function.

7. On the `greengrassObjectClassification` configuration page, choose Edit.
8. On the Group-specific Lambda configuration page, use the following values:
   - For Memory limit, enter 96 MB.
   - For Timeout, enter 10 seconds.
   - For Lambda lifecycle, choose Make this function long-lived and keep it running indefinitely.
   - For Read access to /sys directory, choose Enable.

For more information, see the section called “Lifecycle Configuration” (p. 187).
9. Under Environment variables, create a key-value pair. A key-value pair is required by functions that interact with MXNet models on a Raspberry Pi.

For the key, use MXNET_ENGINE_TYPE. For the value, use NaiveEngine.

**Note**

In your own user-defined Lambda functions, you can optionally set the environment variable in your function code.

10. Keep the default values for all other properties and choose **Update**.

**Step 6: Add Resources to the Greengrass Group**

In this step, you create resources for the camera module and the ML inference model. You also affiliate the resources with the Lambda function, which makes it possible for the function to access the resources on the core device.

First, create two local device resources for the camera: one for shared memory and one for the device interface. For more information about local resource access, see Access Local Resources with Lambda Functions and Connectors (p. 200).

1. On the group configuration page, choose **Resources**.
2. On the **Local** tab, choose **Add a local resource**.
3. On the **Create a local resource** page, use the following values:
   - For **Resource name**, enter `videoCoreSharedMemory`.
   - For **Resource type**, choose **Device**.
   - For **Device path**, enter `/dev/vcsm`.

   The device path is the local absolute path of the device resource. This path can only refer to a character device or block device under `/dev`.

   - For **Group owner file access permission**, choose **Automatically add OS group permissions of the Linux group that owns the resource**.

   The **Group owner file access permission** option lets you grant additional file access permissions to the Lambda process. For more information, see Group Owner File Access Permission (p. 201).
4. Under **Lambda function affiliations**, choose **Select**.

5. Choose **greengrassObjectClassification**, choose **Read and write access**, and then choose **Done**.

Next, you add a local device resource for the camera interface.

6. Choose **Add another resource**.

7. On the **Create a local resource** page, use the following values:

   - For **Resource name**, enter **videoCoreInterface**.
   - For **Resource type**, choose **Device**.
   - For **Device path**, enter /dev/vchiq.
   - For **Group owner file access permission**, choose **Automatically add OS group permissions of the Linux group that owns the resource**.


8. Under **Lambda function affiliations**, choose **Select**.
9. Choose **greengrassObjectClassification**, choose **Read and write access**, and then choose **Done**.
10. At the bottom of the page, choose **Save**.

Now, add the inference model as a machine learning resource. This step includes uploading the `squeezenet.zip` model package to Amazon S3.

1. On the **Resources** page for your group, choose **Machine Learning**, and then choose **Add a machine learning resource**.
2. On the **Create a machine learning resource** page, for **Resource name**, enter **squeezenet_model**.

3. For **Model source**, choose **Upload a model in S3**.
4. Under **Model from S3**, choose **Select**.
5. Choose **Upload a model**. This opens up a new tab to the Amazon S3 console.
6. In the Amazon S3 console tab, upload the `squeezenet.zip` file to an Amazon S3 bucket. For information, see **How Do I Upload Files and Folders to an S3 Bucket?**
Note
For the bucket to be accessible, your bucket name must contain the string greengrass. Choose a unique name (such as greengrass-bucket-user-id-epoch-time). Don't use a period (.) in the bucket name.

7. In the AWS IoT Greengrass console tab, locate and choose your Amazon S3 bucket. Locate your uploaded squeezenet.zip file, and choose Select. You might need to choose Refresh to update the list of available buckets and files.

8. For Local path, enter /greengrass-machine-learning/mxnet/squeezenet.

   This is the destination for the local model in the Lambda runtime namespace. When you deploy the group, AWS IoT Greengrass retrieves the source model package and then extracts the contents to the specified directory. The sample Lambda function for this tutorial is already configured to use this path (in the model_path variable).

9. Under Lambda function affiliations, choose Select.
10. Choose greengrassObjectClassification, choose Read-only access, and then choose Done.
11. Choose Save.

Using Amazon SageMaker Trained Models

This tutorial uses a model that's stored in Amazon S3, but you can easily use Amazon SageMaker models too. The AWS IoT Greengrass console has built-in Amazon SageMaker integration, so you don't need to manually upload these models to Amazon S3. For requirements and limitations for using Amazon SageMaker models, see the section called “Supported Model Sources” (p. 222).

To use an Amazon SageMaker model:

- For Model source, choose Use an existing SageMaker model, and then choose the name of the model's training job.
- For Local path, enter the path to the directory where your Lambda function looks for the model.

Step 7: Add a Subscription to the Greengrass Group

In this step, you add a subscription to the group. This subscription enables the Lambda function to send prediction results to AWS IoT by publishing to an MQTT topic.

1. On the group configuration page, choose Subscriptions, and then choose Add Subscription.

2. On the Select your source and target page, configure the source and target, as follows:
   a. In Select a source, choose Lambdas, and then choose greengrassObjectClassification.
   b. In Select a target, choose Services, and then choose IoT Cloud.
c. Choose Next.

3. On the Filter your data with a topic page, in Topic filter, enter hello/world, and then choose Next.


Step 8: Deploy the Greengrass Group

In this step, you deploy the current version of the group definition to the Greengrass core device. The definition contains the Lambda function, resources, and subscription configurations that you added.

1. Make sure that the AWS IoT Greengrass core is running. Run the following commands in your Raspberry Pi terminal, as needed.

   a. To check whether the daemon is running:

   ```
   ps aux | grep -E 'greengrass.*daemon'
   ```

   If the output contains a root entry for /greengrass/ggc/packages/1.9.4/bin/daemon, then the daemon is running.

   **Note**
   The version in the path depends on the AWS IoT Greengrass Core software version that's installed on your core device.

   b. To start the daemon:

   ```
   cd /greengrass/ggc/core/
   sudo ./greengrassd start
   ```
2. On the group configuration page, choose **Deployments**, and from the **Actions** menu, choose **Deploy**.

<table>
<thead>
<tr>
<th>Deployments</th>
<th>Group history overview</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subscriptions</td>
<td>There are no deployments for this Greengrass Group yet</td>
</tr>
<tr>
<td>Cores</td>
<td></td>
</tr>
<tr>
<td>Devices</td>
<td></td>
</tr>
<tr>
<td>Lambdas</td>
<td></td>
</tr>
</tbody>
</table>

3. On the **Configure how devices discover your core** page, choose **Automatic detection**.

This enables devices to automatically acquire connectivity information for the core, such as IP address, DNS, and port number. Automatic detection is recommended, but AWS IoT Greengrass also supports manually specified endpoints. You're only prompted for the discovery method the first time that the group is deployed.

- Automatically detect Core endpoints (recommended)
  - **Greengrass will detect and override connection information as it changes.**

- Manually configure Core endpoints
  - **Manually manage connection information. This can be accessed via your Core device's settings.**

**Note**
If prompted, grant permission to create the **Greengrass service role (p. 443)** and associate it with your AWS account in the current AWS Region. This role allows AWS IoT Greengrass to access your resources in AWS services.

The **Deployments** page shows the deployment timestamp, version ID, and status. When completed, the status displayed for the deployment should be **Successfully completed**.

For troubleshooting help, see *Troubleshooting (p. 516).*

## Test the Inference App

Now you can verify whether the deployment is configured correctly. To test, you subscribe to the `hello/world` topic and view the prediction results that are published by the Lambda function.

**Note**
If a monitor is attached to the Raspberry Pi, the live camera feed is displayed in a preview window.

1. In the AWS IoT console, choose **Test**.
2. For **Subscriptions**, use the following values:

   - For the subscription topic, use `hello/world`.
   - For **MQTT payload display**, choose **Display payloads as strings**.

3. Choose **Subscribe to topic**.

   If the test is successful, the messages from the Lambda function appear at the bottom of the page. Each message contains the top five prediction results of the image, using the format: probability, predicted class ID, and corresponding class name.
Troubleshooting AWS IoT Greengrass ML Inference

If the test is not successful, you can try the following troubleshooting steps. Run the commands in your Raspberry Pi terminal.

Check Error Logs

1. Switch to the root user and navigate to the log directory. Access to AWS IoT Greengrass logs requires root permissions.

   ```
   sudo su
   cd /greengrass/ggc/var/log
   ```

2. Check `runtime.log` or `python_runtime.log`.

   For more information, see the section called “Troubleshooting with Logs” (p. 534).

Unpacking Error in Runtime.log

If `runtime.log` contains an error similar to the following, make sure that your tar.gz source model package has a parent directory.

```
Greengrass deployment error: unable to download the artifact model-arn: Error while processing.
Error while unpacking the file from /tmp/greengrass/artifacts/model-arn/path to /greengrass/ggc/deployment/path/model-arn/squeezenet/squeezenet_v1.1-0000.params: no such file or directory
```

If your package doesn't have a parent directory that contains the model files, use the following command to repackage the model:

```
tar -zcvf model.tar.gz ./model
```

For example:

```#
tar -zcvf test.tar.gz ./test
./test
./test/some.file
./test/some.file2
./test/some.file3
```

**Note**

Don't include trailing /* characters in this command.

Verify That the Lambda Function Is Successfully Deployed

1. List the contents of the deployed Lambda in the /lambda directory. Replace the placeholder values before you run the command.

   ```
   ls -la
   ```
2. Verify that the directory contains the same content as the greengrassObjectClassification.zip deployment package that you uploaded in Step 4: Create and Publish a Lambda Function (p. 228).

Make sure that the .py files and dependencies are in the root of the directory.

Verify That the Inference Model Is Successfully Deployed

1. Find the process identification number (PID) of the Lambda runtime process:

   ```
   ps aux | grep lambda-function-name
   ```

   In the output, the PID appears in the second column of the line for the Lambda runtime process.

2. Enter the Lambda runtime namespace. Be sure to replace the placeholder `pid` value before you run the command.

   **Note**
   
   This directory and its contents are in the Lambda runtime namespace, so they aren't visible in a regular Linux namespace.

   ```
   sudo nsenter -t pid -m /bin/bash
   ```

3. List the contents of the local directory that you specified for the ML resource.

   ```
   cd /greengrass-machine-learning/mxnet/squeezenet/
   ls -ls
   ```

   You should see the following files:

   ```
   32 -rw-r--r-- 1 ggc_user ggc_group 31675 Nov 18 15:19 synset.txt
   32 -rw-r--r-- 1 ggc_user ggc_group 28707 Nov 18 15:19 squeezenet_v1.1-symbol.json
   4832 -rw-r--r-- 1 ggc_user ggc_group 4945062 Nov 18 15:19 squeezenet_v1.1-0000.params
   ```

Next Steps

Next, explore other inference apps. AWS IoT Greengrass provides other Lambda functions that you can use to try out local inference. You can find the examples package in the precompiled libraries folder that you downloaded in the section called “Install the MXNet Framework” (p. 227).

Configuring an NVIDIA Jetson TX2

To run this tutorial on an NVIDIA Jetson TX2, you provide source images and configure the Lambda function. If you're using the GPU, you must also add local device resources.

To learn how to configure your Jetson so you can install the AWS IoT Greengrass Core software, see the section called “Setting Up Other Devices” (p. 75).

1. Download static PNG or JPG images for the Lambda function to use for image classification. The app works best with small image files. Alternatively, you can instrument a camera on the Jetson board to capture the source images.
Save your image files in the directory that contains the greengrassObjectClassification.py file (or in a subdirectory of this directory). This is in the Lambda function deployment package that you upload in the section called “Create and Publish a Lambda Function” (p. 228).

2. Edit the configuration of the Lambda function to increase the Memory limit value. Use 500 MB for CPU, or 2048 MB for GPU. Follow the procedure in the section called “Add the Lambda Function to the Group” (p. 232).

3. GPU only: Add the following local device resources. Follow the procedure in the section called “Add Resources to the Group” (p. 234).

   For each resource:
   
   - For Resource type, choose Device.
   - For Group owner file access permission, choose Automatically add OS group permissions of the Linux group that owns the resource.
   - For Lambda function affiliations, grant Read and write access to your Lambda function.

<table>
<thead>
<tr>
<th>Name</th>
<th>Device path</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvhost-ctrl</td>
<td>/dev/nvhost-ctrl</td>
</tr>
<tr>
<td>nvhost-gpu</td>
<td>/dev/nvhost-gpu</td>
</tr>
<tr>
<td>nvhost-ctrl-gpu</td>
<td>/dev/nvhost-ctrl-gpu</td>
</tr>
<tr>
<td>nvhost-dbq-gpu</td>
<td>/dev/nvhost-dbq-gpu</td>
</tr>
<tr>
<td>nvhost-prof-gpu</td>
<td>/dev/nvhost-prof-gpu</td>
</tr>
<tr>
<td>nvmap</td>
<td>/dev/nvmap</td>
</tr>
</tbody>
</table>

How to Configure Optimized Machine Learning Inference Using the AWS Management Console

To follow the steps in this tutorial, you must be using AWS IoT Greengrass Core v1.6 or later.

You can use the Amazon SageMaker Neo deep learning compiler to optimize the prediction efficiency of native machine learning inference models in many frameworks. You can then download the optimized model and install the Amazon SageMaker Neo deep learning runtime and deploy them to your AWS IoT Greengrass devices for faster inference.

This tutorial describes how to use the AWS Management Console to configure a Greengrass group to run a Lambda inference example that recognizes images from a camera locally, without sending data to the cloud. The inference example accesses the camera module on a Raspberry Pi. In this tutorial, you download a prepackaged model that is trained by Resnet-50 and optimized in the Neo deep learning compiler. You then use the model to perform local image classification on your AWS IoT Greengrass device.

The tutorial contains the following high-level steps:

1. Configure the Rasberry Pi (p. 245).
2. Install the Amazon SageMaker Neo deep learning runtime (p. 246).
3. Create an inference Lambda function (p. 246).
4. Add the Lambda function to the group (p. 250).
5. Add the Amazon SageMaker Neo optimized model resource to the group (p. 251).
6. Add the camera device resource to the group (p. 252).
7. Add subscriptions to the group (p. 254).
8. Deploy the group (p. 255).

**Prerequisites**

To complete this tutorial, you need:

- Raspberry Pi 4 Model B, or Raspberry Pi 3 Model B/B+, set up and configured for use with AWS IoT Greengrass. To learn how to set up your Raspberry Pi with AWS IoT Greengrass, see Module 1 and Module 2 of Getting Started with AWS IoT Greengrass (p. 64).

  **Note**
  
  The Raspberry Pi might require a 2.5A power supply to run the deep learning frameworks that are typically used for image classification. A power supply with a lower rating might cause the device to reboot.

- 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.

- A Greengrass group and a Greengrass core. To learn how to create a Greengrass group or core, see Getting Started with AWS IoT Greengrass (p. 64).

  **Note**
  
  This tutorial uses a Raspberry Pi, but AWS IoT Greengrass supports other platforms, such as Intel Atom (p. 258) and NVIDIA Jetson TX2 (p. 259).

**Step 1: Configure the Raspberry Pi**

In this step, you install updates to the Raspbian operating system, install the camera module software and Python dependencies, and enable the camera interface.

Run the following commands in your Raspberry Pi terminal.

1. Install updates to Raspbian.

   ```
   sudo apt-get update
   sudo apt-get dist-upgrade
   ```

2. Install the picamera interface for the camera module and other Python libraries that are required for this tutorial.

   ```
   sudo apt-get install -y python-dev python-setuptools python-pip python-picamera
   ```

3. Reboot the Raspberry Pi.

   ```
   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. Use the following command to test the camera setup.

   ```bash
   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.

---

**Step 2: Install the Amazon SageMaker Neo Deep Learning Runtime**

In this step, you download the Neo deep learning runtime and install it onto your Raspberry Pi.


2. Transfer the downloaded `dlr-1.0-py2-armv7l.tar.gz` file from your computer to your Raspberry Pi. You can also use the following `scp` command with a path to save your file, such as `/home/pi/`:

   ```bash
   scp dlr-1.0-py2-armv7l.tar.gz pi@your-device-ip-address:path-to-save-file
   ```

3. Use the following commands to remotely sign in to your Raspberry Pi and extract the installer files.

   ```bash
   ssh pi@your-device-ip-address
cd path-to-save-file
tar -xvzf dlr-1.0-py2-armv7l.tar.gz
   ```

4. Install the Neo deep learning runtime.

   ```bash
   cd dlr-1.0-py2-armv7l/
   chmod 755 install-dlr.sh
   sudo ./install-dlr.sh
   ```

   This package contains an `examples` directory that contains several files you use to run this tutorial. This directory also contains version 1.2.0 of the AWS IoT Greengrass Core SDK for Python. You can also download the latest version of the SDK from the [AWS IoT Greengrass Core SDK](https://aws.amazon.com/greengrass/downloads) downloads page.

---

**Step 3: Create an Inference Lambda Function**

In this step, you create a deployment package and a Lambda function that is configured to use the deployment package. Then, you publish a function version and create an alias.

1. On your computer, unzip the downloaded `dlr-1.0-py2-armv7l.tar.gz` file you previously copied to your Raspberry Pi.

   ```bash
   cd path-to-downloaded-runtime
tar -xvzf dlr-1.0-py2-armv7l.tar.gz
   ```
2. The resulting `dlr-1.0-py2-armv7l` directory contains an `examples` folder. It contains `inference.py`, the example code used in this tutorial for inference. You can view this code as a usage example to create your own inference code.

Compress the files in the `examples` folder into a file named `optimizedImageClassification.zip`.

**Note**
When you create the .zip file, verify that the .py files and dependencies are in the root of the directory.

```
cd path-to-downloaded-runtime/dlr-1.0-py2-armv7l/examples
zip -r optimizedImageClassification.zip .
```

This .zip file is your deployment package. This package contains the function code and dependencies, including the code example that invokes the Neo deep learning runtime Python APIs to perform inference with the Neo deep learning compiler models. You upload this deployment package later.

3. Now, create the Lambda function.

   In the AWS IoT console, in the navigation pane, choose **Greengrass**, and then choose **Groups**.

4. Choose the Greengrass group where you want to add the Lambda function.

5. On the group configuration page, choose **Lambdas**, and then choose **Add Lambda**.
6. On the **Add a Lambda to your Greengrass Group** page, choose **Create new Lambda**. This opens the AWS Lambda console.

![Add Lambda button]

6. On the **Add a Lambda to your Greengrass Group** page, choose **Create new Lambda**. This opens the AWS Lambda console.

7. Choose **Author from scratch** and use the following values to create your function:

   - For **Function name**, enter **optimizedImageClassification**.
   - For **Runtime**, choose **Python 2.7**.

   For **Permissions**, keep the default setting. This creates an execution role that grants basic Lambda permissions. This role isn't used by AWS IoT Greengrass.

8. Choose **Create function**.

Now, upload your Lambda function deployment package and register the handler.
1. On the Configuration tab for the optimizedImageClassification function, for Function code, use the following values:
   - For Code entry type, choose Upload a .zip file.
   - For Runtime, choose Python 2.7.
   - For Handler, enter inference.handler.
2. Choose Upload.
3. Choose your optimizedImageClassification.zip deployment package.
4. Choose Save.

Next, publish the first version of your Lambda function. Then, create an alias for the version.

**Note**
Greengrass groups can reference a Lambda function by alias (recommended) or by version. Using an alias makes it easier to manage code updates because you don't have to change your subscription table or group definition when the function code is updated. Instead, you just point the alias to the new function version.

1. From the Actions menu, choose Publish new version.
2. For Version description, enter First version, and then choose Publish.
3. On the optimizedImageClassification: 1 configuration page, from the Actions menu, choose Create alias.
4. On the Create a new alias page, use the following values:
   - For Name, enter mlTestOpt.
   - For Version, enter 1.

**Note**
AWS IoT Greengrass doesn't support Lambda aliases for $LATEST versions.
5. Choose **Create**.

Now, add the Lambda function to your Greengrass group.

**Step 4: Add the Lambda Function to the Greengrass Group**

In this step, you add the Lambda function to the group, and then configure its lifecycle.

First, add the Lambda function to your Greengrass group.

1. On the **Add a Lambda to your Greengrass Group** page, choose **Use existing Lambda**.

2. Choose **optimizedImageClassification**, and then choose **Next**.

3. On the **Select a Lambda version** page, choose **Alias:mlTestOpt**, and then choose **Finish**.

Next, configure the lifecycle of the Lambda function.

1. On the **Lambdas** page, choose the **optimizedImageClassification** Lambda function.

2. On the **optimizedImageClassification** configuration page, choose **Edit**.

3. On the **Group-specific Lambda configuration** page, use the following values:
   - For **Memory limit**, enter **1024 MB**.
   - For **Timeout**, enter **10 seconds**.
   - For **Lambda lifecycle**, choose **Make this function long-lived and keep it running indefinitely**.
   - For **Read access to /sys directory**, choose **Enable**.

   For more information, see the section called “Lifecycle Configuration” (p. 187).

4. Choose **Update**.
Step 5: Add a Amazon SageMaker Neo-Optimized Model Resource to the Greengrass Group

In this step, you create a resource for the optimized ML inference model and upload it to an Amazon S3 bucket. Then, you locate the Amazon S3 uploaded model in the AWS IoT Greengrass console and affiliate the newly created resource with the Lambda function. This makes it possible for the function to access its resources on the core device.

1. On your computer, navigate to the Neo deep learning runtime installer package that you unpacked earlier. Navigate to the resnet50 directory.

   ```bash
   cd path-to-downloaded-runtime/dlr-1.0-py2-armv7l/models/resnet50
   ```

   This directory contains precompiled model artifacts for an image classification model trained with Resnet-50. Compress the files inside the resnet50 directory to create resnet50.zip.

   ```bash
   zip -r resnet50.zip .
   ```

2. On the group configuration page for your AWS IoT Greengrass group, choose Resources. Navigate to the Machine Learning section and choose Add machine learning resource. On the Create a machine learning resource page, for Resource name, enter resnet50_model.

3. For Model source, choose Upload a model in S3.

4. Under Model from S3, choose Select.

   **Note**
   Currently, optimized Amazon SageMaker models are stored automatically in Amazon S3. You can find your optimized model in your Amazon S3 bucket using this option. For more information about model optimization in Amazon SageMaker, see the Amazon SageMaker Neo documentation.

5. Choose Upload a model.

6. On the Amazon S3 console tab, upload your zip file to an Amazon S3 bucket. For information, see How Do I Upload Files and Folders to an S3 Bucket?

   **Note**
   Your bucket name must contain the string `greengrass`. Choose a unique name (such as `greengrass-dlr-bucket-user-id-epoch-time`). Don't use a period (.) in the bucket name.

7. In the AWS IoT Greengrass console tab, locate and choose your Amazon S3 bucket. Locate your uploaded resnet50.zip file, and choose Select. You might need to refresh the page to update the list of available buckets and files.
8. In **Local path**, enter `/ml_model`.

   ![Local path](image)

   This is the destination for the local model in the Lambda runtime namespace. When you deploy the group, AWS IoT Greengrass retrieves the source model package and then extracts the contents to the specified directory.

   **Note**
   We strongly recommend that you use the exact path provided for your local path. Using a different local model destination path in this step causes some troubleshooting commands provided in this tutorial to be inaccurate. If you use a different path, you must set up a `MODEL_PATH` environment variable that uses the exact path you provide here. For information about environment variables, see [AWS Lambda Environment Variables](#).

9. Under **Lambda function affiliations**, choose **Select**.
10. Choose `optimizedImageClassification`, choose **Read-only access**, and then choose **Done**.
11. Choose **Save**.

---

### Step 6: Add Your Camera Device Resource to the Greengrass Group

In this step, you create a resource for the camera module and affiliate it with the Lambda function, allowing the resource to be accessible on the AWS IoT Greengrass core.

1. On the group configuration page, choose **Resources**.

   - Deployments
   - Subscriptions
   - Cores
   - Devices
   - Lambdas
   - **Resources**
   - Connectors
   - Tags
   - Settings

2. On the **Local** tab, choose **Add local resource**.
3. On the Create a local resource page, use the following values:
   - For Resource name, enter videoCoreSharedMemory.
   - For Resource type, choose Device.
   - For Device path, enter /dev/vcs.".

   The device path is the local absolute path of the device resource. This path can refer only to a character device or block device under /dev.
   - For Group owner file access permission, choose Automatically add OS group permissions of the Linux group that owns the resource.

   The Group owner file access permission option lets you grant additional file access permissions to the Lambda process. For more information, see Group Owner File Access Permission (p. 201).

4. Under Lambda function affiliations, choose Select.
5. Choose optimizedImageClassification, choose Read and write access, and then choose Done.

6. At the bottom of the page, choose Add another resource.
7. On the Create a local resource page, use the following values:
   - For Resource name, enter videoCoreInterface.
   - For Resource type, choose Device.
• For **device path**, enter `/dev/vchiq`.

• For **Group owner file access permission**, choose **Automatically add OS group permissions of the Linux group that owns the resource**.

Add a new local resource

<table>
<thead>
<tr>
<th>Resource name</th>
<th>videoCoreInterface</th>
</tr>
</thead>
</table>

**Resource type**

- Device
- Volume

**Device path**

- /dev/vchiq

**Group owner file access permission**

- An AWS IoT Greengrass Lambda function process normally runs without an OS Group. However, you can give additional file access permissions to the Lambda function process.

- **No OS group**
- **Automatically add OS group permissions of the Linux group that owns the resource**
- **Specify another OS group to add permission**

8. Under **Lambda function affiliations**, choose Select.

9. Choose **optimizedImageClassification**, choose **Read and write access**, and then choose **Done**.

10. Choose **Save**.

### Step 7: Add Subscriptions to the Greengrass Group

In this step, you add subscriptions to the group. These subscriptions enable the Lambda function to send prediction results to AWS IoT by publishing to an MQTT topic.

1. On the group configuration page, choose **Subscriptions**, and then choose **Add Subscription**.

2. On the **Select your source and target** page, configure the source and target, as follows:
   a. In **Select a source**, choose **Lambdas**, and then choose **optimizedImageClassification**.
   b. In **Select a target**, choose **Services**, and then choose **IoT Cloud**.
   c. Choose **Next**.
3. On the Filter your data with a topic page, in Optional topic filter, enter /resnet-50/predictions, and then choose Next.


5. Add a second subscription. On the Select your source and target page, configure the source and target, as follows:
   a. In Select a source, choose Services, and then choose IoT Cloud.
   b. In Select a target, choose Lambdas, and then choose optimizedImageClassification.
   c. Choose Next.

6. On the Filter your data with a topic page, in Optional topic filter, enter /resnet-50/test, and then choose Next.

7. Choose Finish.

Step 8: Deploy the Greengrass Group

In this step, you deploy the current version of the group definition to the Greengrass core device. The definition contains the Lambda function, resources, and subscription configurations that you added.

1. Make sure that the AWS IoT Greengrass core is running. Run the following commands in your Raspberry Pi terminal, as needed.
   a. To check whether the daemon is running:

   ```bash
   ps aux | grep -E 'greengrass.*daemon'
   ```

   If the output contains a root entry for /greengrass/ggc/packages/latest-core-version/bin/daemon, then the daemon is running.
b. To start the daemon:

```
  cd /greengrass/ggc/core/
  sudo ./greengrassd start
```

2. On the group configuration page, choose **Deployments**, and from the **Actions** menu, choose **Deploy**.

3. On the **Configure how devices discover your core** page, choose **Automatic detection**.

   This enables devices to automatically acquire connectivity information for the core, such as IP address, DNS, and port number. Automatic detection is recommended, but AWS IoT Greengrass also supports manually specified endpoints. You're only prompted for the discovery method the first time that the group is deployed.

   **Note**
   
   If prompted, grant permission to create the Greengrass service role (p. 443) and associate it with your AWS account in the current AWS Region. This role allows AWS IoT Greengrass to access your resources in AWS services.

   The **Deployments** page shows the deployment timestamp, version ID, and status. When completed, the status displayed for the deployment should be **Successfully completed**.

   **For troubleshooting help, see Troubleshooting (p. 516).**
Test the Inference Example

Now you can verify whether the deployment is configured correctly. To test, you subscribe to the /resnet-50/predictions topic and publish any message to the /resnet-50/test topic. This triggers the Lambda function to take a photo with your Raspberry Pi and perform inference on the image it captures.

**Note**
If a monitor is attached to the Raspberry Pi, the live camera feed is displayed in a preview window.

1. On the AWS IoT console home page, choose Test.

2. For Subscriptions, choose Subscribe to a Topic. Use the following values. Leave the remaining options at their defaults.
   - For Subscription topic, enter /resnet-50/predictions.
   - For MQTT payload display, choose Display payloads as strings.

3. Choose Subscribe to topic.

4. On the /resnet-50/predictions page, specify the /resnet-50/test topic to publish to. Choose Publish to topic.

5. If the test is successful, the published message causes the Raspberry Pi camera to capture an image. A message from the Lambda function appears at the bottom of the page. This message contains the prediction result of the image, using the format: predicted class name, probability, and peak memory usage.
Configuring an Intel Atom

To run this tutorial on an Intel Atom device, you provide source images and configure the Lambda function. To use the GPU for inference, you must have OpenCL version 1.0 or later installed on your device. You must also add a local device resource.

1. Download static PNG or JPG images for the Lambda function to use for image classification. The example works best with small image files.

   Save your image files in the directory that contains the `inference.py` file (or in a subdirectory of this directory). This is in the Lambda function deployment package that you upload in the section called "Create an Inference Lambda Function" (p. 246).

   **Note**
   If you are using AWS DeepLens, you can choose to instead use the onboard camera or mount your own camera to capture images and perform inference on them. However, we strongly recommend you start with static images first.

2. Edit the configuration of the Lambda function. Follow the procedure in the section called "Add the Lambda Function to the Group" (p. 250).

   a. Increase the **Memory limit** value to 3000 MB.
   b. Increase the **Timeout** value to 2 minutes. This ensures that the request does not time out too early. It takes a few minutes after setup to run inference.
   c. For **Read access to /sys directory**, choose **Enable**.
   d. For **Lambda lifecycle**, choose **Make this function long-lived and keep it running indefinitely**.

3. Add the required local device resource.

   a. On the group configuration page, choose **Resources**.
b. On the **Local** tab, choose **Add a local resource**.

c. Define the resource:

   - For **Resource name**, enter `renderD128`.
   - For **Resource type**, choose **Device**.
   - For **Device path**, enter `/dev/dri/renderD128`.
   - For **Group owner file access permission**, choose **Automatically add OS group permissions of the Linux group that owns the resource**.
   - For **Lambda function affiliations**, grant **Read and write access** to your Lambda function.

### Configuring an NVIDIA Jetson TX2

To run this tutorial on an NVIDIA Jetson TX2, you provide source images and configure the Lambda function. To use the GPU for inference, you must install CUDA 9.0 and cuDNN 7.0 on your device when you image your board with Jetpack 3.3. You must also add local device resources.

To learn how to configure your Jetson so you can install the AWS IoT Greengrass Core software, see the section called “Setting Up Other Devices” (p. 75).

1. Download static PNG or JPG images for the Lambda function to use for image classification. The example works best with small image files.

   Save your image files in the directory that contains the `inference.py` file (or in a subdirectory of this directory). This is in the Lambda function deployment package that you upload in the section called “Create an Inference Lambda Function” (p. 246).

   **Note**
   
   You can instead choose to instrument a camera on the Jetson board to capture the source images. However, we strongly recommend you start with static images first.

2. Edit the configuration of the Lambda function. Follow the procedure in the section called “Add the Lambda Function to the Group” (p. 250).
a. Increase the **Memory limit** value. To use the provided model in GPU mode, use 2048 MB.
b. Increase the **Timeout** value to 5 minutes. This ensures that the request does not time out too early. It takes a few minutes after setup to run inference.
c. For **Lambda lifecycle**, choose **Make this function long-lived and keep it running indefinitely**.
d. For **Read access to /sys directory**, choose **Enable**.

3. Add the required local device resources.
   a. On the group configuration page, choose **Resources**.

   Deployments
   Subscriptions
   Cores
   Devices
   Lambdas
   **Resources**
   Connectors
   Tags
   Settings
   
   b. On the **Local** tab, choose **Add a local resource**.
   c. Define each resource:
      - For **Resource name** and **Device path**, use the values in the following table. Create one device resource for each row in the table.
      - For **Resource type**, choose **Device**.
      - For **Group owner file access permission**, choose **Automatically add OS group permissions of the Linux group that owns the resource**.
      - For **Lambda function affiliations**, grant **Read and write access** to your Lambda function.

<table>
<thead>
<tr>
<th>Name</th>
<th>Device path</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvhost-ctrl</td>
<td>/dev/nvhost-ctrl</td>
</tr>
<tr>
<td>nvhost-gpu</td>
<td>/dev/nvhost-gpu</td>
</tr>
<tr>
<td>nvhost-ctrl-gpu</td>
<td>/dev/nvhost-ctrl-gpu</td>
</tr>
<tr>
<td>nvhost-db-gpu</td>
<td>/dev/nvhost-db-gpu</td>
</tr>
<tr>
<td>nvhost-prof-gpu</td>
<td>/dev/nvhost-prof-gpu</td>
</tr>
</tbody>
</table>

260
<table>
<thead>
<tr>
<th>Name</th>
<th>Device path</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvmap</td>
<td>/dev/nvmap</td>
</tr>
</tbody>
</table>

### Troubleshooting AWS IoT Greengrass ML Inference

If the test is not successful, you can try the following troubleshooting steps. Run the commands in your Raspberry Pi terminal.

#### Check error logs

1. Switch to the root user and navigate to the log directory. Access to AWS IoT Greengrass logs requires root permissions.

   ```
   sudo su
   cd /greengrass/ggc/var/log
   ```

2. Check `runtime.log` for any errors.

   ```
   cat system/runtime.log | grep 'ERROR'
   ```

   You can also look in your user-defined Lambda function log for any errors:

   ```
   cat user/your-region/your-account-id/lambda-function-name.log | grep 'ERROR'
   ```

   For more information, see the section called “Troubleshooting with Logs” (p. 534).

#### Verify the Lambda function is successfully deployed

1. List the contents of the deployed Lambda in the `/lambda` directory. Replace the placeholder values before you run the command.

   ```
   ls -la
   ```

2. Verify that the directory contains the same content as the `optimizedImageClassification.zip` deployment package that you uploaded in Step 3: Create an Inference Lambda Function (p. 246).

   Make sure that the `.py` files and dependencies are in the root of the directory.

#### Verify the inference model is successfully deployed

1. Find the process identification number (PID) of the Lambda runtime process:

   ```
   ps aux | grep lambda-function-name
   ```

   In the output, the PID appears in the second column of the line for the Lambda runtime process.
2. Enter the Lambda runtime namespace. Be sure to replace the placeholder `pid` value before you run the command.

   **Note**
   This directory and its contents are in the Lambda runtime namespace, so they aren’t visible in a regular Linux namespace.

   ```
sudo nsenter -t pid -m /bin/bash
   ```

3. List the contents of the local directory that you specified for the ML resource.

   **Note**
   If your ML resource path is something other than `ml_model`, you must substitute that here.

   ```
cd /ml_model
ls -ls
   ```

You should see the following files:

<table>
<thead>
<tr>
<th>Permissions</th>
<th>Owner</th>
<th>Group</th>
<th>Size (Bytes)</th>
<th>Date</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>56 -rw-r--r--</td>
<td>ggc_user</td>
<td>ggc_group</td>
<td>56703</td>
<td>Oct 29 20:07</td>
<td>model.json</td>
</tr>
<tr>
<td>196152 -rw-r--r--</td>
<td>ggc_user</td>
<td>ggc_group</td>
<td>200855043</td>
<td>Oct 29 20:08</td>
<td>model.params</td>
</tr>
<tr>
<td>256 -rw-r--r--</td>
<td>ggc_user</td>
<td>ggc_group</td>
<td>261848</td>
<td>Oct 29 20:07</td>
<td>model.so</td>
</tr>
<tr>
<td>32 -rw-r--r--</td>
<td>ggc_user</td>
<td>ggc_group</td>
<td>30564</td>
<td>Oct 29 20:08</td>
<td>synset.txt</td>
</tr>
</tbody>
</table>

**Lambda function cannot find /dev/dri/renderD128**

This can occur if OpenCL cannot connect to the GPU devices it needs. You must create device resources for the necessary devices for your Lambda function.

**Next Steps**

Next, explore other optimized models. For information, see the [Amazon SageMaker Neo documentation](https://aws.amazon.com/sagemaker-neo/).
Deploy Secrets to the AWS IoT Greengrass Core

This feature is available for AWS IoT Greengrass Core v1.7 and later.

AWS IoT Greengrass lets you authenticate with services and applications from Greengrass devices without hard-coding passwords, tokens, or other secrets.

AWS Secrets Manager is a service that you can use to securely store and manage your secrets in the cloud. AWS IoT Greengrass extends Secrets Manager to Greengrass core devices, so your connectors (p. 283) and Lambda functions can use local secrets to interact with services and applications. For example, the Twilio Notifications connector uses a locally stored authentication token.

To integrate a secret into a Greengrass group, you create a group resource that references the Secrets Manager secret. This secret resource references the cloud secret by ARN. To learn how to create, manage, and use secret resources, see the section called “Work with Secret Resources” (p. 267).

AWS IoT Greengrass encrypts your secrets while in transit and at rest. During group deployment, AWS IoT Greengrass fetches the secret from Secrets Manager and creates a local, encrypted copy on the Greengrass core. After you rotate your cloud secrets in Secrets Manager, redeploy the group to propagate the updated values to the core.

The following diagram shows the high-level process of deploying a secret to the core. Secrets are encrypted in transit and at rest.

Using AWS IoT Greengrass to store your secrets locally offers these advantages:

- **Decoupled from code (not hard-coded).** This supports centrally managed credentials and helps protect sensitive data from the risk of compromise.
- **Available for offline scenarios.** Connectors and functions can securely access local services and software when disconnected from the internet.
- **Controlled access to secrets.** Only authorized connectors and functions in the group can access your secrets. AWS IoT Greengrass uses private key encryption to secure your secrets. Secrets are encrypted in transit and at rest. For more information, see the section called “Secrets Encryption” (p. 264).
• **Controlled rotation.** After you rotate your secrets in Secrets Manager, redeploy the Greengrass group to update the local copies of your secrets. For more information, see the section called “Creating and Managing Secrets” (p. 267).

  **Important**
  AWS IoT Greengrass doesn't automatically update the values of local secrets after cloud versions are rotated. To update local values, you must redeploy the group.

**Secrets Encryption**

AWS IoT Greengrass encrypts secrets in transit and at rest.

  **Important**
  Make sure that your user-defined Lambda functions handle secrets securely and don't log any sensitive data that's stored in the secret. For more information, see Mitigate the Risks of Logging and Debugging Your Lambda Function in the AWS Secrets Manager User Guide. Although this documentation specifically refers to rotation functions, the recommendation also applies to Greengrass Lambda functions.

**Encryption in transit**

AWS IoT Greengrass uses Transport Layer Security (TLS) to encrypt all communication over the internet and local network. This protects secrets while in transit, which occurs when secrets are retrieved from Secrets Manager and deployed to the core. For supported TLS cipher suites, see the section called “TLS Cipher Suites Support” (p. 442).

**Encryption at rest**

AWS IoT Greengrass uses the private key specified in `config.json` (p. 24) for encryption of the secrets that are stored on the core. For this reason, secure storage of the private key is critical for protecting local secrets. In the AWS shared responsibility model, it's the responsibility of the customer to guarantee secure storage of the private key on the core device.

AWS IoT Greengrass supports two modes of private key storage:

- Using file system permissions (default). The private key is used to secure the data key, which is used to encrypt local secrets. The data key is rotated with each group deployment. The AWS IoT Greengrass core is the only entity that has access to the private key. Greengrass connectors or Lambda functions that are affiliated with a secret resource get the value of the secret from the core.

**Requirements**

These are the requirements for local secret support:

- You must be using AWS IoT Greengrass Core v1.7 or later.
• To get the values of local secrets, your user-defined Lambda functions must use AWS IoT Greengrass Core SDK v1.3.0 or later.

• The private key used for local secrets encryption must be specified in the Greengrass configuration file. By default, AWS IoT Greengrass uses the core private key stored in the file system. To provide your own private key, see the section called “Specify the Private Key for Secret Encryption” (p. 265). Only the RSA key type is supported.

  Note
  Currently, AWS IoT Greengrass supports only the PKCS#1 v1.5 padding mechanism for encryption and decryption of local secrets when using hardware-based private keys. If you're following vendor-provided instructions to manually generate hardware-based private keys, make sure to choose PKCS#1 v1.5. AWS IoT Greengrass doesn't support Optimal Asymmetric Encryption Padding (OAEP).

• AWS IoT Greengrass must be granted permission to get your secret values. This allows AWS IoT Greengrass to fetch the values during group deployment. If you're using the default Greengrass service role, then AWS IoT Greengrass already has access to secrets with names that start with greengrass-. To customize access, see the section called “Allow AWS IoT Greengrass to Get Secret Values” (p. 266).

  Note
  We recommend that you use this naming convention to identify the secrets that AWS IoT Greengrass is allowed to access, even if you customize permissions. The console uses different permissions to read your secrets, so it's possible that you can select secrets in the console that AWS IoT Greengrass doesn't have permission to fetch. Using a naming convention can help avoid a permission conflict, which results in a deployment error.

Specify the Private Key for Secret Encryption

In this procedure, you provide the path to a private key that's used for local secret encryption. This must be an RSA key with a minimum length of 2048 bits. For more information about private keys used on the AWS IoT Greengrass core, see the section called “Security Principals” (p. 439).

AWS IoT Greengrass supports two modes of private key storage: hardware-based or file system-based (default). For more information, see the section called “Secrets Encryption” (p. 264).

Follow this procedure only if you want to change the default configuration, which uses the core private key in the file system. These steps are written with the assumption that you created your group and core as described in Module 2 (p. 76) of the Getting Started tutorial.

1. Open the `config.json` file that's located in the `/greengrass-root/config` directory.

   Note
   `greengrass-root` represents the path where the AWS IoT Greengrass Core software is installed on your device. If you installed the software by following the steps in the Getting Started (p. 64) tutorial, then this is the `/greengrass` directory.

2. In the `crypto.principals.SecretsManager` object, for the `privateKeyPath` property, enter the path of the private key:

   • If your private key is stored in the file system, specify the absolute path to the key. For example:

   ```json
   "SecretsManager" : {
     "privateKeyPath" : "file:///somepath/hash.private.key"
   }
   ```

   • If your private key is stored in a hardware security module (HSM), specify the path using the RFC 7512 PKCS#11 URI scheme. For example:

   ```json
   "SecretsManager" : {
   ```

265
Allow AWS IoT Greengrass to Get Secret Values

In this procedure, you add an inline policy to the Greengrass service role that allows AWS IoT Greengrass to get the values of your secrets.

Follow this procedure only if you want to grant AWS IoT Greengrass custom permissions to your secrets or if your Greengrass service role doesn’t include the AWSGreengrassResourceAccessRolePolicy managed policy. AWSGreengrassResourceAccessRolePolicy grants access to secrets with names that start with `greengrass-`.

1. Run the following CLI command to get the ARN of the Greengrass service role:

   ```bash
   aws greengrass get-service-role-for-account --region region
   ```

   The returned ARN contains the role name.

   ```json
   {
   "AssociatedAt": "time-stamp",
   "RoleArn": "arn:aws:iam::account-id:role/service-role/role-name"
   }
   ```

   You use the ARN or name in the following step.

2. Add an inline policy that allows the secretsmanager:GetSecretValue action. For instructions, see Adding and Removing IAM Policies in the IAM User Guide.

   You can grant granular access by explicitly listing secrets or using a wildcard * naming scheme, or you can grant conditional access to versioned or tagged secrets. For example, the following policy allows AWS IoT Greengrass to read only the specified secrets.

   ```json
   {
   "Version": "2012-10-17",
   "Statement": [
   {
   "Effect": "Allow",
   "Action": [
   "secretsmanager:GetSecretValue"
   ],
   "Resource": [
   ]
   }
   ]
   ```

For more information, see the section called “Hardware Security Configuration” (p. 450).

Note
Currently, AWS IoT Greengrass supports only the PKCS#1 v1.5 padding mechanism for encryption and decryption of local secrets when using hardware-based private keys. If you’re following vendor-provided instructions to manually generate hardware-based private keys, make sure to choose PKCS#1 v1.5. AWS IoT Greengrass doesn’t support Optimal Asymmetric Encryption Padding (OAEP).
Note
If you use a customer-managed AWS KMS key to encrypt secrets, your Greengrass service role must also allow the kms:Decrypt action.

For more information about IAM policies for Secrets Manager, see Authentication and Access Control for AWS Secrets Manager and Actions, Resources, and Context Keys You Can Use in an IAM Policy or Secret Policy for AWS Secrets Manager in the AWS Secrets Manager User Guide.

See Also

• What Is AWS Secrets Manager? in the AWS Secrets Manager User Guide
• PKCS #1: RSA Encryption Version 1.5

Working with Secret Resources

AWS IoT Greengrass uses secret resources to integrate secrets from AWS Secrets Manager into a Greengrass group. A secret resource is a reference to a Secrets Manager secret. For more information, see Deploy Secrets to the Core (p. 263).

On the AWS IoT Greengrass core device, connectors and Lambda functions can use the secret resource to authenticate with services and applications, without hard-coding passwords, tokens, or other credentials.

Creating and Managing Secrets

In a Greengrass group, a secret resource references the ARN of a Secrets Manager secret. When the secret resource is deployed to the core, the value of the secret is encrypted and made available to affiliated connectors and Lambda functions. For more information, see the section called “Secrets Encryption” (p. 264).

You use Secrets Manager to create and manage the cloud versions of your secrets. You use AWS IoT Greengrass to create, manage, and deploy your secret resources.

Important
We recommend that you follow the best practice of rotating your secrets in Secrets Manager. Then, deploy the Greengrass group to update the local copies of your secrets. For more information, see Rotating Your AWS Secrets Manager Secrets in the AWS Secrets Manager User Guide.

To make a secret available on the Greengrass core

1. Create a secret in Secrets Manager. This is the cloud version of your secret, which is centrally stored and managed in Secrets Manager. Management tasks include rotating secret values and applying resource policies.

2. Create a secret resource in AWS IoT Greengrass. This is a type of group resource that references the cloud secret by ARN. You can reference a secret only once per group.

3. Configure your connector or Lambda function. You must affiliate the resource with a connector or function by specifying corresponding parameters or properties. This allows them to get the value of the locally deployed secret resource. For more information, see the section called “Using Local Secrets” (p. 270).
4. Deploy the Greengrass group. During deployment, AWS IoT Greengrass fetches the value of the cloud secret and creates (or updates) the local secret on the core.

Secrets Manager logs an event in AWS CloudTrail each time that AWS IoT Greengrass retrieves a secret value. AWS IoT Greengrass doesn't log any events related to the deployment or usage of local secrets. For more information about Secrets Manager logging, see Monitor the Use of Your AWS Secrets Manager Secrets in the AWS Secrets Manager User Guide.

Including Staging Labels in Secret Resources

Secrets Manager uses staging labels to identify specific versions of a secret value. Staging labels can be system-defined or user-defined. Secrets Manager assigns the AWSCURRENT label to the most recent version of the secret value. Staging labels are commonly used to manage secrets rotation. For more information about Secrets Manager versioning, see Key Terms and Concepts for AWS Secrets Manager in the AWS Secrets Manager User Guide.

Secret resources always include the AWSCURRENT staging label, and they can optionally include other staging labels if they're required by a Lambda function or connector. During group deployment, AWS IoT Greengrass retrieves the values of the staging labels that are referenced in the group, and then creates or updates the corresponding values on the core.

Create and Manage Secret Resources (Console)

Creating Secret Resources (Console)

In the AWS IoT Greengrass console, you create and manage secret resources from the Secrets tab on the group's Resources page. For tutorials that create a secret resource and add it to a group, see the section called “How To Create a Secret Resource (Console)” (p. 272) and the section called “Get Started with Connectors (Console)” (p. 403).

Managing Secret Resources (Console)

Management tasks for the secret resources in your Greengrass group include adding secret resources to the group, removing secret resources from the group, and changing the set of staging labels (p. 268) that are included in a secret resource.

If you point to a different secret from Secrets Manager, you must also edit any connectors that use the secret:
1. On the group configuration page, choose Connectors.
2. From the connector's contextual menu, choose Edit.
3. The Edit parameters page displays a message to inform you that the secret ARN changed. To confirm the change, choose Save.

If you delete a secret in Secrets Manager, remove the corresponding secret resource from the group and from connectors and Lambda functions that reference it. Otherwise, during group deployment, AWS IoT Greengrass returns an error that the secret can’t be found. Also update your Lambda function code as needed.

Create and Manage Secret Resources (CLI)

Creating Secret Resources (CLI)

In the AWS IoT Greengrass API, a secret is a type of group resource. The following example creates a resource definition with an initial version that includes a secret resource named MySecretResource. For a tutorial that creates a secret resource and adds it to a group version, see the section called “Get Started with Connectors (CLI)” (p. 413).

The secret resource references the ARN of the corresponding Secrets Manager secret and includes two staging labels in addition to AWSCURRENT, which is always included.

```bash
aws greengrass create-resource-definition --name MyGreengrassResources --initial-version '{
  "Resources": [
    {
      "Id": "my-resource-id",
      "Name": "MySecretResource",
      "ResourceDataContainer": {
        "SecretsManagerSecretResourceData": {
          "AdditionalStagingLabelsToDownload": [
            "Label1",
            "Label2"
          ]
        }
      }
    }
  ]
}
```

Managing Secret Resources (CLI)

Management tasks for the secret resources in your Greengrass group include adding secret resources to the group, removing secret resources from the group, and changing the set of staging labels (p. 268) that are included in a secret resource.

In the AWS IoT Greengrass API, these changes are implemented by using versions.

The AWS IoT Greengrass API uses versions to manage groups. Versions are immutable, so to add or change group components—for example, the group's devices, functions, and resources—you must create versions of new or updated components. Then, you create and deploy a group version that contains the target version of each component. To learn more about groups, see the section called “AWS IoT Greengrass Groups” (p. 6).

For example, to change the set of staging labels for a secret resource:

1. Create a resource definition version that contains the updated secret resource. The following example adds a third staging label to the secret resource from the previous section.
Note
To add more resources to the version, include them in the Resources array.

```bash
aws greengrass create-resource-definition --name MyGreengrassResources --initial-version
'{
   "Resources": [
     {
       "Id": "my-resource-id",
       "Name": "MySecretResource",
       "ResourceDataContainer": {
         "SecretsManagerSecretResourceData": {
           "AdditionalStagingLabelsToDownload": [
             "Label1",
             "Label2",
             "Label3"
           ]
         }
       }
     }
   ]
}'
```

2. If the ID of the secret resource is changed, update connectors and functions that use the secret resource. In the new versions, update the parameter or property that corresponds to the resource ID. If the ARN of the secret is changed, you must also update the corresponding parameter for any connectors that use the secret.

   Note
   The resource ID is an arbitrary identifier that's provided by the customer.

3. Create a group version that contains the target version of each component that you want to send to the core.

4. Deploy the group version.

For a tutorial that shows how to create and deploy secret resources, connectors, and functions, see the section called “Get Started with Connectors (CLI)” (p. 413).

If you delete a secret in Secrets Manager, remove the corresponding secret resource from the group and from connectors and Lambda functions that reference it. Otherwise, during group deployment, AWS IoT Greengrass returns an error that the secret can't be found. Also update your Lambda function code as needed. You can remove a local secret by deploying a resource definition version that doesn't contain the corresponding secret resource.

Using Local Secrets in Connectors and Lambda Functions

Greengrass connectors and Lambda functions use local secrets to interact with services and applications. The `AWS_CURRENT` value is used by default, but values for other staging labels (p. 268) included in the secret resource are also available.

Connectors and functions must be configured before they can access local secrets. This affiliates the secret resource with connector or function.

Connectors

If a connector requires access to a local secret, it provides parameters that you configure with the information it needs to access the secret.
Using Local Secrets

- To learn how to do this in the AWS IoT Greengrass console, see the section called “Get Started with Connectors (Console)” (p. 403).
- To learn how to do this with the AWS IoT Greengrass CLI, see the section called “Get Started with Connectors (CLI)” (p. 413).

For information about requirements for individual connectors, see the section called “AWS-Provided Greengrass Connectors” (p. 288).

The logic for accessing and using the secret is built into the connector.

**Lambda functions**

To allow a Greengrass Lambda function to access a local secret, you configure the function's properties.

- To learn how to do this in the AWS IoT Greengrass console, see the section called “How To Create a Secret Resource (Console)” (p. 272).
- To do this in the AWS IoT Greengrass API, you provide the following information in the ResourceAccessPolicies property.
  - **ResourceId**: The ID of the secret resource in the Greengrass group. This is the resource that references the ARN of the corresponding Secrets Manager secret.
  - **Permission**: The type of access that the function has to the resource. Only `ro` (read-only) permission is supported for secret resources.

The following example creates a Lambda function that can access the MyApiKey secret resource.

```bash
aws greengrass create-function-definition --name MyGreengrassFunctions --initial-version '{
  "Functions": [
    {
      "Id": "MyLambdaFunction",
      "FunctionConfiguration": {
        "Pinned": false,
        "MemorySize": 16384,
        "Timeout": 10,
        "Environment": {
          "ResourceAccessPolicies": [
            {
              "ResourceId": "MyApiKey",
              "Permission": "ro"
            }
          ],
          "AccessSysfs": true
        }
      }
    }
  ]
}'
```

To access local secrets at runtime, Greengrass Lambda functions call the `get_secret_value` function from the `secretsmanager` client in the AWS IoT Greengrass Core SDK (v1.3.0 or later).

The following example shows how to use the AWS IoT Greengrass Core SDK for Python to get a secret. It passes the name of the secret to the `get_secret_value` function. `SecretId` can be the name or ARN of the Secrets Manager secret (not the secret resource).

```python
import greengrasssdk
```

14

271
# Creating a Greengrass Core SDK client
```python
client = greengrasssdk.client('secretsmanager')
```

# This handler is called when the function is invoked
# It uses the secretsmanager client to get the value of a secret
```python
def function_handler(event, context):
    response = client.get_secret_value(SecretId='greengrass-MySecret-abc')
    raw_secret = response.get('SecretString')
```

For text type secrets, the `get_secret_value` function returns a string. For binary type secrets, it returns a base64-encoded string.

**Important**
Make sure that your user-defined Lambda functions handle secrets securely and don't log any any sensitive data that's stored in the secret. For more information, see Mitigate the Risks of Logging and Debugging Your Lambda Function in the AWS Secrets Manager User Guide. Although this documentation specifically refers to rotation functions, the recommendation also applies to Greengrass Lambda functions.

The current value of the secret is returned by default. This is the version that the AWSCURRENT staging label is attached to. To access a different version, pass the name of the corresponding staging label for the optional `VersionStage` argument. For example:

```python
import greengrasssdk
```

```python
# Creating a greengrass core sdk client
client = greengrasssdk.client('secretsmanager')

# This handler is called when the function is invoked
# It uses the secretsmanager client to get the value of a specific secret version
```python
def function_handler(event, context):
    response = client.get_secret_value(SecretId='greengrass-MySecret-abc',
                                        VersionStage='MyTargetLabel')
    raw_secret = response.get('SecretString')
```

For another example function that calls `get_secret_value`, see Create a Lambda Function Deployment Package (p. 276).

## How To Create a Secret Resource (Console)

This feature is available for AWS IoT Greengrass Core v1.7 and later.

This tutorial shows how to use the AWS Management Console to add a secret resource to a Greengrass group. A secret resource is a reference to a secret from AWS Secrets Manager. For more information, see Deploy Secrets to the Core (p. 263).

On the AWS IoT Greengrass core device, connectors and Lambda functions can use the secret resource to authenticate with services and applications, without hard-coding passwords, tokens, or other credentials.

In this tutorial, you start by creating a secret in the AWS Secrets Manager console. Then, in the AWS IoT Greengrass console, you add a secret resource to a Greengrass group from the group's Resources page. This secret resource references the Secrets Manager secret. Later, you attach the secret resource to a Lambda function, which allows the function to get the value of the local secret.

**Note**
Alternatively, the console allows you to create a secret and secret resource when you configure a connector or Lambda function. You can do this from the connector's Configure parameters page or the Lambda function's Resources page.
Only connectors that contain parameters for secrets can access secrets. For a tutorial that shows how the Twilio Notifications connector uses a locally stored authentication token, see the section called “Get Started with Connectors (Console)” (p. 403).

The tutorial contains the following high-level steps:

1. Create a Secrets Manager Secret (p. 273)
2. Add a Secret Resource to a Group (p. 274)
3. Create a Lambda Function Deployment Package (p. 276)
4. Create a Lambda Function (p. 277)
5. Add the Function to the Group (p. 278)
6. Attach the Secret Resource to the Function (p. 278)
7. Add Subscriptions to the Group (p. 279)
8. Deploy the Group (p. 280)

The tutorial should take about 20 minutes to complete.

Prerequisites

To complete this tutorial, you need:

- A Greengrass group and a Greengrass core (v1.7 or later). To learn how to create a Greengrass group and core, see Getting Started with AWS IoT Greengrass (p. 64). The Getting Started tutorial also includes steps for installing the AWS IoT Greengrass Core software.

- AWS IoT Greengrass must be configured to support local secrets. For more information, see Secrets Requirements (p. 264).

  **Note**
  This includes allowing access to your Secrets Manager secrets. If you're using the default Greengrass service role, Greengrass has permission to get the values of secrets with names that start with `greengrass-`.

- To get the values of local secrets, your user-defined Lambda functions must use AWS IoT Greengrass Core SDK v1.3.0 or later.

Step 1: Create a Secrets Manager Secret

In this step, you use the AWS Secrets Manager console to create a secret.

1. Sign in to the AWS Secrets Manager console.

   **Note**
   For more information about this process, see Step 1: Create and Store Your Secret in AWS Secrets Manager in the AWS Secrets Manager User Guide.

2. Choose Store a new secret.
3. Under Select secret type, choose Other type of secrets.
4. Under Specify the key-value pairs to be stored for this secret:
   - For Key, enter test.
   - For Value, enter abedefghi.
5. Keep DefaultEncryptionKey selected for the encryption key, and then choose Next.

   Note
   You aren't charged by AWS KMS if you use the default AWS managed key that Secrets Manager creates in your account.

6. For Secret name, enter greengrass-TestSecret, and then choose Next.

   Note
   By default, the Greengrass service role allows AWS IoT Greengrass to get the value of secrets with names that start with greengrass-. For more information, see secrets requirements (p. 264).

7. This tutorial doesn't require rotation, so choose Disable automatic rotation, and then choose Next.

8. On the Review page, review your settings, and then choose Store.

   Next, you create a secret resource in your Greengrass group that references the secret.

**Step 2: Add a Secret Resource to a Greengrass Group**

In this step, you configure a group resource that references the Secrets Manager secret.

1. In the AWS IoT console, choose Greengrass, and then choose Groups.

2. Choose the group that you want to add the secret resource to.

3. On the group configuration page, choose Resources, and then choose Secret. This tab displays the secret resources that belong to the group. You can add, edit, and remove secret resources from this tab.
4. Choose **Add a secret resource**.

5. On the **Add a secret resource to your group** page, choose **Select**, and then choose `greengrass-TestSecret`.

6. On the **Select labels (Optional)** page, choose **Next**. The AWSCURRENT staging label represents the latest version of the secret. This label is always included in a secret resource.

   **Note**
   This tutorial requires the AWSCURRENT label only. You can optionally include labels that are required by your Lambda function or connector.

7. On the **Name your secret resource** page, enter `MyTestSecret`, and then choose **Save**.
Step 3: Create a Lambda Function Deployment Package

To create a Lambda function, you must first create a Lambda function deployment package that contains the function code and dependencies. Greengrass Lambda functions require the AWS IoT Greengrass Core SDK (p. 175) for tasks such as communicating with MQTT messages in the core environment and accessing local secrets. This tutorial creates a Python function, so you use the Python version of the SDK in the deployment package.

**Note**
To get the values of local secrets, your user-defined Lambda functions must use AWS IoT Greengrass Core SDK v1.3.0 or later.

1. Download the AWS IoT Greengrass Core SDK for Python from the AWS IoT Greengrass Core SDK (p. 22) downloads page.
2. Unzip the downloaded package to get the SDK. The SDK is the greengrasssdk folder.
3. Save the following Python code function in a local file named secret_test.py.

```python
import greengrasssdk

# Create SDK clients.
secrets_client = greengrasssdk.client('secretsmanager')
message_client = greengrasssdk.client('iot-data')
message = ''

# This handler is called when the function is invoked.
# It uses the 'secretsmanager' client to get the value of the test secret using the secret name.
# The test secret is a text type, so the SDK returns a string.
# For binary secret values, the SDK returns a base64-encoded string.
def function_handler(event, context):
    response = secrets_client.get_secret_value(SecretId='greengrass-TestSecret')
    secret_value = response.get('SecretString')
    if secret_value is None:
        message = 'Failed to retrieve secret.'
    else:
        message = 'Success! Retrieved secret.'

    message_client.publish(topic='secrets/output', payload=message)
    print('published: ' + message)
```

The `get_secret_value` function supports the name or ARN of the Secrets Manager secret for the `SecretId` value. This example uses the secret name. For this example secret, AWS IoT Greengrass returns the key-value pair: `{"test":"abcdefghi"}`.

**Important**
Make sure that your user-defined Lambda functions handle secrets securely and don't log any any sensitive data that's stored in the secret. For more information, see Mitigate the Risks of Logging and Debugging Your Lambda Function in the AWS Secrets Manager User Guide. Although this documentation specifically refers to rotation functions, the recommendation also applies to Greengrass Lambda functions.

4. Zip the following items into a file named secret_test_python.zip. When you create the ZIP file, include only the code and dependencies, not the containing folder.

   - `secret_test.py`. App logic.
   - `greengrasssdk`. Required library for all Python Greengrass Lambda functions.
This is your Lambda function deployment package.

Step 4: Create a Lambda Function

In this step, you use the AWS Lambda console to create a Lambda function and configure it to use your deployment package. Then, you publish a function version and create an alias.

1. First, create the Lambda function.
   a. In the AWS Management Console, choose Services, and open the AWS Lambda console.
   b. Choose Create function and then choose Author from scratch.
   c. In the Basic information section, use the following values:
      • For Function name, enter SecretTest.
      • For Runtime, choose Python 2.7.
      • For Permissions, keep the default setting. This creates an execution role that grants basic Lambda permissions. This role isn’t used by AWS IoT Greengrass.
   d. At the bottom of the page, choose Create function.

2. Next, register the handler and upload your Lambda function deployment package.
   a. On the Configuration tab for the SecretTest function, in Function code, use the following values:
      • For Code entry type, choose Upload a .zip file.
      • For Runtime, choose Python 2.7.
      • For Handler, enter secret_test.function_handler
   b. Choose Upload.
   c. Choose your secret_test_python.zip deployment package.
   d. Choose Save.
   Note
   The Test button on the AWS Lambda console doesn't work with this function. The AWS IoT Greengrass Core SDK doesn't contain modules that are required to run your Greengrass Lambda functions independently in the AWS Lambda console. These modules (for example, greengrass_common) are supplied to the functions after they are deployed to your Greengrass core.

   Tip
   You can see your code in the Function code section by choosing Edit code inline from the Code entry type menu.

3. Now, publish the first version of your Lambda function and create an alias for the version.
   Note
   Greengrass groups can reference a Lambda function by alias (recommended) or by version. Using an alias makes it easier to manage code updates because you don't have to change your subscription table or group definition when the function code is updated. Instead, you just point the alias to the new function version.
   a. From the Actions menu, choose Publish new version.
   b. For Version description, enter First version, and then choose Publish.
   c. On the SecretTest: 1 configuration page, from the Actions menu, choose Create alias.
   d. On the Create a new alias page, use the following values:
Add the Function to the Group

- For **Name**, enter `GG_SecretTest`.
- For **Version**, choose 1.

**Note**
AWS IoT Greengrass doesn’t support Lambda aliases for $LATEST$ versions.

Choose **Create**.

Now you’re ready to add the Lambda function to your Greengrass group and attach the secret resource.

**Step 5: Add the Lambda Function to the Greengrass Group**

In this step, you add the Lambda function to the Greengrass group in the AWS IoT console.

1. On the group configuration page, choose **Lambdas**, and then choose **Add Lambda**.

2. On the **Add a Lambda to your Greengrass Group** page, choose **Use existing Lambda**.

3. On the **Use existing Lambda** page, choose **SecretTest**, and then choose **Next**.

4. On the **Select a Lambda version** page, choose **Alias:GG_SecretTest**, and then choose **Finish**.

Next, affiliate the secret resource with the function.

**Step 6: Attach the Secret Resource to the Lambda Function**

In this step, you attach the secret resource to the Lambda function in your Greengrass group. This affiliates the resource with the function, which allows the function to get the value of the local secret.

1. On the group’s **Lambdas** page, choose the **SecretTest** function.
2. On the function's details page, choose Resources, choose Secret, and then choose Attach a secret resource.

3. On the Attach a secret resource to your Lambda function page, choose Choose secret resource.

4. On the Select a secret resource from your group page, choose MyTestSecret, and then choose Save.

Step 7: Add Subscriptions to the Greengrass Group

In this step, you add subscriptions that allow AWS IoT and the Lambda function to exchange messages. One subscription allows AWS IoT to invoke the function, and one allows the function to send output data to AWS IoT.

1. On the group configuration page, choose Subscriptions, and then choose Add Subscription.

2. Create a subscription that allows AWS IoT to publish messages to the function.

   On the Select your source and target page, configure the source and target:
   
   a. For Select a source, choose Services, and then choose IoT Cloud.
   
   b. For Select a target, choose Lambdas, and then choose SecretTest.
   
   c. Choose Next.

3. On the Filter your data with a topic page, for Topic filter, enter secrets/input, and then choose Next.


5. Repeat steps 1 - 4 to create a subscription that allows the function to publish status to AWS IoT.

   a. For Select a source, choose Lambdas, and then choose SecretTest.
b. For **Select a target**, choose **Services**, and then choose **IoT Cloud**.

c. For **Topic filter**, enter **secrets/output**.

### Step 8: Deploy the Greengrass Group

Deploy the group to the core device. During deployment, AWS IoT Greengrass fetches the value of the secret from Secrets Manager and creates a local, encrypted copy on the core.

1. Make sure that the AWS IoT Greengrass core is running. Run the following commands in your Raspberry Pi terminal, as needed.
   a. To check whether the daemon is running:

   ```
   ps aux | grep -E 'greengrass.*daemon'
   ```

   If the output contains a root entry for `/greengrass/ggc/packages/ggc-version/bin/daemon`, then the daemon is running.

   **Note**
   The version in the path depends on the AWS IoT Greengrass Core software version that's installed on your core device.

   b. To start the daemon:

   ```
   cd /greengrass/ggc/core/
   sudo ./greengrassd start
   ```

2. On the group configuration page, choose **Deployments**, and from the **Actions** menu, choose **Deploy**.

3. On the **Configure how devices discover your core** page, choose **Automatic detection**.

   This enables devices to automatically acquire connectivity information for the core, such as IP address, DNS, and port number. Automatic detection is recommended, but AWS IoT Greengrass also supports manually specified endpoints. You're only prompted for the discovery method the first time that the group is deployed.
Note
If prompted, grant permission to create the Greengrass service role (p. 443) and associate it with your AWS account in the current AWS Region. This role allows AWS IoT Greengrass to access your resources in AWS services.

The Deployments page shows the deployment timestamp, version ID, and status. When completed, the deployment should show a Successfully completed status.

For troubleshooting help, see Troubleshooting (p. 516).

Test the Function
1. On the AWS IoT console home page, choose Test.

2. For Subscriptions, use the following values, and then choose Subscribe to topic.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subscription topic</td>
<td>secrets/output</td>
</tr>
<tr>
<td>MQTT payload display</td>
<td>Display payloads as strings</td>
</tr>
</tbody>
</table>

3. For Publish, use the following values, and then choose Publish to topic to invoke the function.
Keep the default message. Publishing a message invokes the Lambda function, but the function in this tutorial doesn't process the message body.

If successful, the function publishes a "Success" message.

See Also

- *Deploy Secrets to the Core (p. 263)*
Integrate with Services and Protocols Using Greengrass Connectors

This feature is available for AWS IoT Greengrass Core v1.7 and later.

Greengrass connectors are prebuilt modules that help accelerate the development lifecycle for common edge scenarios. They make it easier to interact with local infrastructure, device protocols, AWS, and other cloud services. With connectors, you can spend less time learning new protocols and APIs and more time focusing on the logic that matters to your business.

The following diagram shows where connectors can fit into the AWS IoT Greengrass landscape.

Many connectors use MQTT messages to communicate with devices and Greengrass Lambda functions in the group, or with AWS IoT and the local shadow service. In the following example, the Twilio Notifications connector receives MQTT messages from a user-defined Lambda function, uses a local reference of a secret from AWS Secrets Manager, and calls the Twilio API.

For tutorials that create this solution, see the section called “Get Started with Connectors (Console)” (p. 403) and the section called “Get Started with Connectors (CLI)” (p. 413).
Greengrass connectors can help you quickly extend device capabilities or create single-purpose devices. Connectors can make it easier to:

- Implement reusable business logic.
- Interact with cloud and local services, including AWS and third-party services.
- Ingest and process device data.
- Enable device-to-device calls using MQTT topic subscriptions and user-defined Lambda functions.

AWS provides a set of Greengrass connectors that simplify interactions with common services and data sources. These prebuilt modules enable scenarios for logging and diagnostics, replenishment, industrial data processing, and alarm and messaging. For more information, see the section called “AWS-Provided Greengrass Connectors” (p. 288).

Requirements

The following requirements apply for connectors:

- You must use AWS IoT Greengrass Core software v1.7 or later.
- You must meet the requirements of each connector that you're using. These requirements might include device prerequisites, required permissions, and limits. For more information, see the section called “AWS-Provided Greengrass Connectors” (p. 288).
- A Greengrass group can contain only one configured instance of a given connector, but the instance can be used in multiple subscriptions. For more information, see the section called “Configuration Parameters” (p. 286).
- Connectors aren’t supported when the Greengrass group is configured to run without containerization. For more information, see the section called “Controlling Greengrass Lambda Function Execution” (p. 178).

Using AWS IoT Greengrass Connectors

A connector is a type of group component. Like other group components, such as devices and user-defined Lambda functions, you add connectors to groups, configure their settings, and deploy them to the AWS IoT Greengrass core. Connectors run in the core environment.

Some connectors can be deployed as simple standalone applications. For example, the Device Defender connector reads system metrics from the core device and sends them to AWS IoT Device Defender for analysis.

Other connectors can be used as building blocks in larger solutions. The following example solution uses the Modbus-RTU Protocol Adapter connector to process messages from sensors and the Twilio Notifications connector to trigger Twilio messages.
Solutions often include user-defined Lambda functions that sit next to connectors and process the data that the connector sends or receives. In this example, the TempMonitor function receives data from Modbus-RTU Protocol Adapter, runs some business logic, and then sends data to Twilio Notifications.

To create and deploy a solution, you follow this general process:

1. Map out the high-level data flow. Identify the data sources, data channels, services, protocols, and resources that you need to work with. In the example solution, this includes data over the Modbus RTU protocol, the physical Modbus serial port, and Twilio.

2. Identify the connectors to include in the solution, and add them to your group. The example solution uses Modbus-RTU Protocol Adapter and Twilio Notifications. To help you find connectors that apply to your scenario, and to learn about their individual requirements, see the section called “AWS-Provided Greengrass Connectors” (p. 288).

3. Identify whether user-defined Lambda functions, devices, or resources are needed, and then create and add them to the group. This might include functions that contain business logic or process data into a format required by another entity in the solution. The example solution uses functions to send Modbus RTU requests and trigger Twilio notifications. It also includes a local device resource for the Modbus RTU serial port and a secret resource for the Twilio authentication token.

   **Note**
   Secret resources reference passwords, tokens, and other secrets from AWS Secrets Manager. Secrets can be used by connectors and Lambda functions to authenticate with services and applications. By default, AWS IoT Greengrass can access secrets with names that start with "greengrass-". For more information, see *Deploy Secrets to the Core* (p. 263).

4. Create subscriptions that allow the entities in the solution to exchange MQTT messages. If a connector is used in a subscription, the connector and the message source or target must use the predefined topic syntax supported by the connector. For more information, see the section called “Inputs and Outputs” (p. 287).

5. Deploy the group to the Greengrass core.

To learn how to create and deploy a connector, see the following tutorials:

- the section called “Get Started with Connectors (Console)” (p. 403)
- the section called “Get Started with Connectors (CLI)” (p. 413)
Configuration Parameters

Many connectors provide parameters that let you customize the behavior or output. These parameters are used during initialization, at runtime, or at other times in the connector lifecycle.

Parameter types and usage vary by connector. For example, the SNS connector has a parameter that configures the default SNS topic, and Device Defender has a parameter that configures the data sampling rate.

A group version can contain multiple connectors, but only one instance of a given connector at a time. This means that each connector in the group can have only one active configuration. However, the connector instance can be used in multiple subscriptions in the group. For example, you can create subscriptions that allow many devices to send data to the Kinesis Firehose connector.

Parameters Used to Access Group Resources

Greengrass connectors use group resources to access the file system, ports, peripherals, and other local resources on the core device. If a connector requires access to a group resource, then it provides related configuration parameters.

Group resources include:

- **Local resources** (p. 200). Directories, files, ports, pins, and peripherals that are present on the Greengrass core device.
- **Machine learning resources** (p. 221). Machine learning models that are trained in the cloud and deployed to the core for local inference.
- **Secret resources** (p. 263). Local, encrypted copies of passwords, keys, tokens, or arbitrary text from AWS Secrets Manager. Connectors can securely access these local secrets and use them to authenticate to services or local infrastructure.

For example, parameters for Device Defender enable access to system metrics in the host `/proc` directory, and parameters for Twilio Notifications enable access to a locally stored Twilio authentication token.

Updating Connector Parameters

Parameters are configured when the connector is added to a Greengrass group. You can change parameter values after the connector is added.

- In the console: From the group configuration page, open **Connectors**, and from the connector's contextual menu, choose **Edit**.

  **Note**
  If the connector uses a secret resource that's later changed to reference a different secret, you must edit the connector's parameters and confirm the change.

- In the API: Create another version of the connector that defines the new configuration.

  The AWS IoT Greengrass API uses versions to manage groups. Versions are immutable, so to add or change group components—for example, the group's devices, functions, and resources—you must create versions of new or updated components. Then, you create and deploy a group version that contains the target version of each component.

  After you make changes to the connector configuration, you must deploy the group to propagate the changes to the core.
Inputs and Outputs

Many Greengrass connectors can communicate with other entities by sending and receiving MQTT messages. MQTT communication is controlled by subscriptions that allow a connector to exchange data with Lambda functions, devices, and other connectors in the Greengrass group, or with AWS IoT and the local shadow service. To allow this communication, you must create subscriptions in the group that the connector belongs to. For more information, see the section called “Greengrass Messaging Workflow” (p. 441).

Connectors can be message publishers, message subscribers, or both. Each connector defines the MQTT topics that it publishes or subscribes to. These predefined topics must be used in the subscriptions where the connector is a message source or message target. For tutorials that include steps for configuring subscriptions for a connector, see the section called “Get Started with Connectors (Console)” (p. 403) and the section called “Get Started with Connectors (CLI)” (p. 413).

**Note**

Many connectors also have built-in modes of communication to interact with cloud or local services. These vary by connector and might require that you configure parameters or add permissions to the group role. For information about connector requirements, see the section called “AWS-Provided Greengrass Connectors” (p. 288).

**Input Topics**

Most connectors receive input data on MQTT topics. Some connectors subscribe to multiple topics for input data. For example, the Serial Stream connector supports two topics:

- `serial/+read/#`
- `serial/+write/#`

For this connector, read and write requests are sent to the corresponding topic. When you create subscriptions, make sure to use the topic that aligns with your implementation.

The `+` and `#` characters in the previous examples are wildcards. These wildcards allow subscribers to receive messages on multiple topics and publishers to customize the topics that they publish to.

- The `+` wildcard can appear anywhere in the topic hierarchy. It can be replaced by one hierarchy item.
  
  As an example, for topic `sensor/+input`, messages can be published to topics `sensor/id-123/input` but not to `sensor/group-a/id-123/input`.
  
  The `#` wildcard can appear only at the end of the topic hierarchy. It can be replaced by zero or more hierarchy items.

  As an example, for topic `sensor/#`, messages can be published to `sensor/`, `sensor/id-123`, and `sensor/group-a/id-123`, but not to `sensor`.

Wildcard characters are valid only when subscribing to topics. Messages can't be published to topics that contain wildcards. Check the documentation for the connector to learn about its input or output topic requirements. For more information, see the section called “AWS-Provided Greengrass Connectors” (p. 288).
Logging

Greengrass connectors contain Lambda functions that write events and errors to Greengrass logs. Depending on your group settings, logs are written to CloudWatch Logs, the local file system, or both. Logs from connectors include the ARN of the corresponding function. The following example ARN is from the Kinesis Firehose connector:

```
arn:aws:lambda:aws-region:account-id:function:KinesisFirehoseClient:1
```

The default logging configuration writes info-level logs to the file system using the following directory structure:

```
greengrass-root/ggc/var/log/user/region/aws/function-name.log
```

For more information about Greengrass logging, see Monitoring (p. 457).

AWS-Provided Greengrass Connectors

AWS provides the following connectors that support common AWS IoT Greengrass scenarios. For more information about how connectors work, see the following documentation:

- Integrate with Services and Protocols Using Connectors (p. 283)
- Get Started with Connectors (Console) (p. 403) or Get Started with Connectors (CLI) (p. 413)

<table>
<thead>
<tr>
<th>Connector</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CloudWatch Metrics (p. 289)</td>
<td>Publishes custom metrics to Amazon CloudWatch.</td>
</tr>
<tr>
<td>Device Defender (p. 296)</td>
<td>Sends system metrics to AWS IoT Device Defender.</td>
</tr>
<tr>
<td>IoT Analytics (p. 299)</td>
<td>Sends data from devices and sensors to AWS IoT Analytics.</td>
</tr>
<tr>
<td>Kinesis Firehose (p. 307)</td>
<td>Sends data to Amazon Kinesis Data Firehose delivery streams.</td>
</tr>
<tr>
<td>ML Feedback (p. 315)</td>
<td>Publishes machine learning model input to the cloud and output to an MQTT topic.</td>
</tr>
<tr>
<td>ML Image Classification (p. 326)</td>
<td>Runs a local image classification inference service. This connector provides versions for several platforms.</td>
</tr>
<tr>
<td>ML Object Detection (p. 343)</td>
<td>Runs a local object detection inference service. This connector provides versions for several platforms.</td>
</tr>
<tr>
<td>Modbus-RTU Protocol Adapter (p. 354)</td>
<td>Sends requests to Modbus RTU devices.</td>
</tr>
<tr>
<td>Raspberry Pi GPIO (p. 365)</td>
<td>Controls GPIO pins on a Raspberry Pi core device.</td>
</tr>
<tr>
<td>Serial Stream (p. 370)</td>
<td>Reads and writes to a serial port on the core device.</td>
</tr>
</tbody>
</table>
CloudWatch Metrics Connector

The CloudWatch Metrics connector (p. 283) publishes custom metrics from Greengrass devices to Amazon CloudWatch. The connector provides a centralized infrastructure for publishing CloudWatch metrics, which you can use to monitor and analyze the Greengrass core environment, and act on local events. For more information, see Using Amazon CloudWatch Metrics in the Amazon CloudWatch User Guide.

This connector receives metric data as MQTT messages. The connector batches metrics that are in the same namespace and publishes them to CloudWatch at regular intervals.

This connector has the following versions.

<table>
<thead>
<tr>
<th>Version</th>
<th>ARN</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>arn:aws:greengrass:region::/connectors/CloudWatchMetrics/versions/2</td>
</tr>
<tr>
<td>1</td>
<td>arn:aws:greengrass:region::/connectors/CloudWatchMetrics/versions/1</td>
</tr>
</tbody>
</table>

For information about version changes, see the Changelog (p. 295).

Requirements

This connector has the following requirements:

- AWS IoT Greengrass Core Software v1.7 or later.
- Python version 2.7 installed on the core device and added to the PATH environment variable.
- An IAM policy added to the Greengrass group role that allows the cloudwatch:PutMetricData action, as shown in the following example.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Stmt1528133056761",
            "Action": [
                "cloudwatch:PutMetricData"
            ],
            "Effect": "Allow",
            "Resource": "*"
        }
    ]
}
```
For more information, see Adding and Removing IAM Policies in the IAM User Guide and Amazon CloudWatch Permissions Reference in the IAM User Guide.

Connector Parameters

This connector provides the following parameters:

**PublishInterval**

The maximum number of seconds to wait before publishing batched metrics for a given namespace. The maximum value is 900. To configure the connector to publish metrics as they are received (without batching), specify 0.

The connector publishes to CloudWatch after it receives 20 metrics in the same namespace or after the specified interval.

*Note*

The connector doesn't guarantee the order of publish events.

Display name in console: **Publish interval**

Required: true

Type: string

Valid values: 0 - 900

Valid pattern: [0-9][1-9]\d|[1-9]\d|900

**PublishRegion**

The AWS Region to post CloudWatch metrics to. This value overrides the default Greengrass metrics region. It is required only when posting cross-region metrics.

Display name in console: **Publish region**

Required: false

Type: string

Valid pattern: ^$|([a-z]{2}-[a-z]+-\d{1})

**MemorySize**

The memory (in KB) to allocate to the connector.

Display name in console: **Memory size**

Required: true

Type: string

Valid pattern: ^[0-9]+$
This limit applies when there’s no connection to the internet and the connector starts to buffer the metrics to publish later. When the buffer is full, the oldest metrics are replaced by new metrics. Metrics in a given namespace are replaced only by metrics in the same namespace.

**Note**
Metrics are not saved if the host process for the connector is interrupted. For example, this can happen during group deployment or when the device restarts.

Display name in console: **Maximum metrics to retain**

Required: true

Type: string

Valid pattern: `^([2-9]\d{3}|[1-9]\d{4,})$`

**Create Connector Example (CLI)**

The following CLI command creates a `ConnectorDefinition` with an initial version that contains the CloudWatch Metrics connector.

```bash
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-version '{
  "Connectors": [
    {
      "Id": "MyCloudWatchMetricsConnector",
      "ConnectorArn": "arn:aws:greengrass:region::/connectors/CloudWatchMetrics/versions/2",
      "Parameters": {
        "PublishInterval": "600",
        "PublishRegion": "us-west-2",
        "MemorySize": "16",
        "MaxMetricsToRetain": "2500"
      }
    }
  ]
}',
```

In the AWS IoT Greengrass console, you can add a connector from the group’s Connectors page. For more information, see the section called “Get Started with Connectors (Console)” (p. 403).

**Input Data**

This connector accepts metrics on an MQTT topic and publishes the metrics to CloudWatch. Input messages must be in JSON format.

**Topic filter**

```
cloudwatch/metric/put
```

**Message properties**

- **request**
  
  Information about 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` API. Only the namespace, metricData.metricName, and metricData.value properties are required.
Required: true
Type: object that includes the following properties:

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/".

Required: true
Type: string
Valid pattern: [^:]*

metricData

The data for the metric.

Required: true
Type: object that includes the following properties:

metricName

The name of the metric.

Required: true
Type: string

dimensions

The dimensions that are associated with the metric. Dimensions provide more information about the metric and its data. A metric can define up to 10 dimensions.

Required: false
Type: array of dimension objects that include the following properties:

name

The dimension name.

Required: false
Type: string

value

The dimension value.

Required: false
Type: string

timestamp

The time that the metric data was received, expressed as the number of milliseconds since Jan 1, 1970 00:00:00 UTC. If this value is omitted, the connector uses the time that it received the message.
Required: false
Type: timestamp
value
The value for the metric.

Note
CloudWatch rejects values that are too small or too large. Values must be in the range of $8.515920 \times 10^{-109}$ to $1.174271 \times 10^{108}$ (Base 10) or $2e^{-360}$ to $2e^{360}$ (Base 2). Special values (for example, NaN, +Infinity, -Infinity) are not supported.

Required: true
Type: double
unit
The unit of the metric.

Required: false
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

Limits
All limits that are imposed by the CloudWatch PutMetricData API apply to metrics when using this connector. The following limits are especially important:

- 40 KB limit on API payload
- 20 metrics per API request
- 150 transactions per second (TPS) for the PutMetricData API

For more information, see CloudWatch Limits in the Amazon CloudWatch User Guide.

Example input

```json
{
  "request": {
    "namespace": "Greengrass",
    "metricData": {
      "metricName": "latency",
      "dimensions": [
        {
          "name": "hostname",
          "value": "test_hostname"
        }
      ],
      "timestamp": 1539027324,
      "value": 123.0,
      "unit": "Seconds"
    }
  }
}
```
Output Data

This connector publishes status information as output data.

**Topic filter**

`cloudwatch/metric/put/status`

**Example output: Success**

The response includes the namespace of the metric data and the `RequestId` field from the CloudWatch response.

```json
{
   "response": {
      "cloudwatch_rid": "70573243-d723-11e8-b095-75ff2EXAMPLE",
      "namespace": "Greengrass",
      "status": "success"
   }
}
```

**Example output: Failure**

```json
{
   "response": {
      "namespace": "Greengrass",
      "error": "InvalidInputException",
      "error_message": "cw metric is invalid",
      "status": "fail"
   }
}
```

*Note*

If the connector detects a retryable error (for example, connection errors), it retries the publish in the next batch.

**Usage Example**

The following example Lambda function sends an input message to the connector.

*Note*

This Python function uses the [*AWS IoT Greengrass Core SDK*](https://docs.aws.amazon.com/greengrass/latest/ug/sdk-py.html) (p. 175) to publish an MQTT message. You can use the following `pip` command to install the Python version of the SDK on your core device:

```
pip install greengrasssdk
```

```python
import greengrasssdk
import time
import json

iot_client = greengrasssdk.client('iot-data')
send_topic = 'cloudwatch/metric/put'
```
def create_request_with_all_fields():
    return {
        "request": {
            "namespace": "Greengrass_CW_Connector",
            "metricData": {
                "metricName": "Count1",
                "dimensions": [
                    {
                        "name": "test",
                        "value": "test"
                    }
                ],
                "value": 1,
                "unit": "Seconds",
                "timestamp": time.time()
            }
        }
    }

def publish_basic_message():
    messageToPublish = create_request_with_all_fields()
    print "Message To Publish: ", messageToPublish
    iot_client.publish(topic=send_topic,
                       payload=json.dumps(messageToPublish))

publish_basic_message()

def function_handler(event, context):
    return

Licenses

The CloudWatch Metrics connector includes the following third-party software/licensing:

- AWS SDK for Python (Boto 3)/Apache 2.0

This connector is released under the Greengrass Core Software License Agreement.

Changelog

The following table describes the changes in each version of the connector.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Fix to reduce excessive logging.</td>
</tr>
<tr>
<td>1</td>
<td>Initial release.</td>
</tr>
</tbody>
</table>

A Greengrass group can contain only one version of the connector at a time.

See Also

- Integrate with Services and Protocols Using Connectors (p. 283)
- the section called “Get Started with Connectors (Console)” (p. 403)
- the section called “Get Started with Connectors (CLI)” (p. 413)
- Using Amazon CloudWatch Metrics in the Amazon CloudWatch User Guide
• **PutMetricData** in the *Amazon CloudWatch API Reference*

## Device Defender Connector

The Device Defender connector (p. 283) notifies administrators of changes in the state of a Greengrass core device. This can help identify unusual behavior that might indicate a compromised device.

This connector reads system metrics from the `/proc` directory on the core device, and then publishes the metrics to AWS IoT Device Defender. For metrics reporting details, see *Device Metrics Document Specification* in the *AWS IoT Developer Guide*.

This connector has the following versions.

<table>
<thead>
<tr>
<th>Version</th>
<th>ARN</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>arn:aws:greengrass:region::/connectors/DeviceDefender/versions/2</td>
</tr>
<tr>
<td>1</td>
<td>arn:aws:greengrass:region::/connectors/DeviceDefender/versions/1</td>
</tr>
</tbody>
</table>

For information about version changes, see the *Changelog* (p. 299).

### Requirements

This connector has the following requirements:

- AWS IoT Greengrass Core Software v1.7 or later.
- Python version 2.7 installed on the core device and added to the PATH environment variable.
- AWS IoT Device Defender configured to use the Detect feature to keep track of violations. For more information, see *Detect* in the *AWS IoT Developer Guide*.
- A local volume resource (p. 200) in the Greengrass group that points to the `/proc` directory. The resource must use the following properties:
  - Source path: `/proc`
  - Destination path: `/host_proc` (or a value that matches the *valid pattern* (p. 297))
  - `AutoAddGroupOwner: true`
- The `psutil` library installed on the AWS IoT Greengrass core. Use the following command to install it:

  ```
  pip install psutil
  ```

- The `cbor` library installed on the AWS IoT Greengrass core. Use the following command to install it:

  ```
  pip install cbor
  ```

### Connector Parameters

This connector provides the following parameters:

**SampleIntervalSeconds**

The number of seconds between each cycle of gathering and reporting metrics. The minimum value is 300 seconds (5 minutes).
Display name in console: **Metrics reporting interval**
Required: true
Type: string

Valid pattern: `^[0-9]*([3][0-9][0-9]|4-9[0-9]{2}|1-9[0-9]{3})$`

**ProcDestinationPath-ResourceId**
The ID of the `/proc` volume resource.

**Note**
This connector is granted read-only access to the resource.

Display name in console: **Resource for /proc directory**
Required: true
Type: string

Valid pattern: `[a-zA-Z0-9_\-]+`

**ProcDestinationPath**
The destination path of the `/proc` volume resource.

Display name in console: **Destination path of /proc resource**
Required: true
Type: string

Valid pattern: `\/[a-zA-Z0-9_\-]+`

### Create Connector Example (CLI)

The following CLI command creates a ConnectorDefinition with an initial version that contains the Device Defender connector.

```bash
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-version
'{
  "Connectors": [
    {
      "Id": "MyDeviceDefenderConnector",
      "ConnectorArn": "arn:aws:greengrass:region::/connectors/DeviceDefender/versions/2",
      "Parameters": {
        "SampleIntervalSeconds": "600",
        "ProcDestinationPath": "/host_proc",
        "ProcDestinationPath-ResourceId": "my-proc-resource"
      }
    }
  ]
}',
```

**Note**
The Lambda function in this connector has a long-lived (p. 187) lifecycle.

In the AWS IoT Greengrass console, you can add a connector from the group's Connectors page. For more information, see the section called “Get Started with Connectors (Console)” (p. 403).
Input Data

This connector doesn't accept MQTT messages as input data.

Output Data

This connector publishes security metrics to AWS IoT Device Defender as output data.

Topic filter

```
$aws/things/+/defender/metrics/json
```

Note

This is the topic syntax that AWS IoT Device Defender expects. The connector replaces the + wildcard with the device name (for example, $aws/things/thing-name/defender/metrics/json).

Example output

For metrics reporting details, see Device Metrics Document Specification in the AWS IoT Developer Guide.

```json
{
  "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
    }
  }
}````
**IoT Analytics**

The IoT Analytics connector sends local device data to AWS IoT Analytics. You can use this connector as a central hub to collect data from sensors on the Greengrass core device and from connected Greengrass devices (p. 8). The connector sends the data to AWS IoT Analytics channels in the current AWS account and Region. It can send data to a default destination channel and to dynamically specified channels.

**Note**

AWS IoT Analytics is a fully managed service that allows you to collect, store, process, and query IoT data. In AWS IoT Analytics, the data can be further analyzed and processed. For example, it can be used to train ML models for monitoring machine health or to test new modeling techniques.

**Licenses**

This connector is released under the Greengrass Core Software License Agreement.

**Changelog**

The following table describes the changes in each version of the connector.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Fix to reduce excessive logging.</td>
</tr>
<tr>
<td>1</td>
<td>Initial release.</td>
</tr>
</tbody>
</table>

A Greengrass group can contain only one version of the connector at a time.

**See Also**

- *Integrate with Services and Protocols Using Connectors* (p. 283)
- the section called “Get Started with Connectors (Console)” (p. 403)
- the section called “Get Started with Connectors (CLI)” (p. 413)
- Device Defender in the *AWS IoT Developer Guide*

---

299
strategies. For more information, see What Is AWS IoT Analytics? in the AWS IoT Analytics User Guide.

The connector accepts formatted and unformatted data on input MQTT topics (p. 303). It supports two predefined topics where the destination channel is specified inline. It can also receive messages on customer-defined topics that are configured in subscriptions (p. 287). This can be used to route messages from devices that publish to fixed topics or handle unstructured or stack-dependent data from resource-constrained devices.

This connector uses the BatchPutMessage API to send data (as a JSON or base64-encoded string) to the destination channel. The connector can process raw data into a format that conforms to API requirements. The connector buffers input messages in per-channel queues and asynchronously processes the batches. It provides parameters that allow you to control queueing and batching behavior and to restrict memory consumption. For example, you can configure the maximum queue size, batch interval, memory size, and number of active channels.

This connector has the following versions.

<table>
<thead>
<tr>
<th>Version</th>
<th>ARN</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>arn:aws:greengrass:region::/connectors/IoTAnalytics/versions/2</td>
</tr>
<tr>
<td>1</td>
<td>arn:aws:greengrass:region::/connectors/IoTAnalytics/versions/1</td>
</tr>
</tbody>
</table>

For information about version changes, see the Changelog (p. 307).

Requirements

This connector has the following requirements:

- AWS IoT Greengrass Core Software v1.7 or later.
- Python version 2.7 installed on the core device and added to the PATH environment variable.
- This connector can be used only in supported AWS Regions. For more information, see the section called “Limits” (p. 306).
- All related AWS IoT Analytics entities (channels, pipeline, datastores, datasets) and workflows are created and configured. For more information, see the AWS CLI or console procedures in the AWS IoT Analytics User Guide.

**Note**

Destination AWS IoT Analytics channels must use the same account and be in the same AWS Region as this connector.

- An IAM policy added to the Greengrass group role that allows the iotanalytics:BatchPutMessage action on destination channels, as shown in the following example. The channels must be in the current AWS account and Region.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt1528133056761",
      "Action": [
        "iotanalytics:BatchPutMessage"
      ],
      "Effect": "Allow",
      "Resource": [...
```
For more information, see Adding and Removing IAM Policies in the IAM User Guide.

Parameters

MemorySize

The amount of memory (in KB) to allocate to this connector.

Display name in console: **Memory size**

Required: true

Type: string

Valid pattern: ^[0-9]+$ $

PublishRegion

The AWS Region that your AWS IoT Analytics channels are created in. Use the same Region as the connector.

**Note**

This must also match the Region for the channels that are specified in the group role (p. 300).

Display name in console: **Publish region**

Required: false

Type: string

Valid pattern: ^$|([a-z]{2}-[a-z]+-\d{1})

PublishInterval

The interval (in seconds) for publishing a batch of received data to AWS IoT Analytics.

Display name in console: **Publish interval**

Required: false

Type: string

Default value: 1

Valid pattern: $|^[0-9]+$ $

IotAnalyticsMaxActiveChannels

The maximum number of AWS IoT Analytics channels that the connector actively watches for. This must be greater than 0, and at least equal to the number of channels that you expect the connector to publish to at a given time.

You can use this parameter to restrict memory consumption by limiting the total number of queues that the connector can manage at a given time. A queue is deleted when all queued messages are sent.
Display name in console: **Maximum number of active channels**
Required: false
Type: string
Default value: 50
Valid pattern: ^$|^[1-9][0-9]*$

*iotAnalyticsQueueDropBehavior*

The behavior for dropping messages from a channel queue when the queue is full.
Display name in console: **Queue drop behavior**
Required: false
Type: string
Valid values: DROP_NEWEST or DROP_OLDEST
Default value: DROP_NEWEST
Valid pattern: ^DROP_NEWEST|DROP_OLDEST$

*iotAnalyticsQueueSizePerChannel*

The maximum number of messages to retain in memory (per channel) before the messages are submitted or dropped. This must be greater than 0.
Display name in console: **Maximum queue size per channel**
Required: false
Type: string
Default value: 2048
Valid pattern: ^$|^[1-9][0-9]*$

*iotAnalyticsBatchSizePerChannel*

The maximum number of messages to send to an AWS IoT Analytics channel in one batch request. This must be greater than 0.
Display name in console: **Maximum number of messages to batch per channel**
Required: false
Type: string
Default value: 5
Valid pattern: ^$|^[1-9][0-9]*$

*iotAnalyticsDefaultChannelName*

The name of the AWS IoT Analytics channel that this connector uses for messages that are sent to a customer-defined input topic.
Display name in console: **Default channel name**
Required: false
Type: string
Valid pattern: ^[a-zA-Z0-9-_.]$

Create Connector Example (CLI)

The following CLI command creates a ConnectorDefinition with an initial version that contains the IoT Analytics connector.

```bash
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-version '{
  "Connectors": [
    {
      "Id": "MyIoTAnalyticsApplication",
      "ConnectorArn": "arn:aws:greengrass:region::/connectors/IoTAnalytics/
versions/2",
      "Parameters": {
        "MemorySize": "65535",
        "PublishRegion": "us-west-1",
        "PublishInterval": "2",
        "IotAnalyticsMaxActiveChannels": "25",
        "IotAnalyticsQueueDropBehavior": "DROP_OLDEST",
        "IotAnalyticsQueueSizePerChannel": "1028",
        "IotAnalyticsBatchSizePerChannel": "5",
        "IotAnalyticsDefaultChannelName": "my_channel"
      }
    }
  ]
}'
```

**Note**

The Lambda function in this connector has a long-lived (p. 187) lifecycle.

In the AWS IoT Greengrass console, you can add a connector from the group's Connectors page. For more information, see the section called “Get Started with Connectors (Console)” (p. 403).

Input Data

This connector accepts data on predefined and customer-defined MQTT topics. Publishers can be Greengrass devices, Lambda functions, or other connectors.

Predefined topics

The connector supports the following two structured MQTT topics that allow publishers to specify the channel name inline.

- A formatted message (p. 304) on the iotanalytics/channels/+messages/put topic. The IoT data in these input messages must be formatted as a JSON or base64-encoded string.
- An unformatted message on the iotanalytics/channels/+messages/binary/put topic. Input messages received on this topic are treated as binary data and can contain any data type.

To publish to predefined topics, replace the + wildcard with the channel name. For example:

```plaintext
iotanalytics/channels/my_channel/messages/put
```

Customer-defined topics

The connector supports the # topic syntax, which allows it to accept input messages on any MQTT topic that you configure in a subscription. We recommend that you specify a topic path instead of
using only the # wildcard in your subscriptions. These messages are sent to the default channel that you specify for the connector.

Input messages on customer-defined topics are treated as binary data. They can use any message format and can contain any data type. You can use customer-defined topics to route messages from devices that publish to fixed topics. You can also use them to accept input data from devices that can't process the data into a formatted message to send to the connector.

For more information about subscriptions and MQTT topics, see the section called “Inputs and Outputs” (p. 287).

The group role must allow the iotanalytics:BatchPutMessage action on all destination channels. For more information, see the section called “Requirements” (p. 300).

**Topic filter:** iotanalytics/channels/+/messages/put

Use this topic to send formatted messages to the connector and dynamically specify a destination channel. This topic also allows you to specify an ID that's returned in the response output. The connector verifies that IDs are unique for each message in the outbound BatchPutMessage request that it sends to AWS IoT Analytics. A message that has a duplicate ID is dropped.

Input data sent to this topic must use the following message format.

**Message properties**

- **request**
  - The data to send to the specified channel.
  - Required: true
  - Type: object that includes the following properties:
    - **message**
      - The device or sensor data as a JSON or base64-encoded string.
      - Required: true
      - Type: string
    - **id**
      - An arbitrary ID for the request. This property is used to map an input request to an output response. When specified, the id property in the response object is set to this value. If you omit this property, the connector generates an ID.
      - Required: false
      - Type: string
      - Valid pattern: .*

**Example input**

```
{  
  "request": {  
    "message": "{"\"temp\":23.33}"
  },  
  "id": "req123"
}
```
**Topic filter:** `iotanalytics/channels/+messages/binary/put`

Use this topic to send unformatted messages to the connector and dynamically specify a destination channel.

The connector data doesn't parse the input messages received on this topic. It treats them as binary data. Before sending the messages to AWS IoT Analytics, the connector encodes and formats them to conform with `BatchPutMessage` API requirements:

- The connector base64-encodes the raw data and includes the encoded payload in an outbound `BatchPutMessage` request.
- The connector generates and assigns an ID to each input message.

**Note**
The connector's response output doesn't include an ID correlation for these input messages.

**Message properties**
None.

**Topic filter:** `#`

Use this topic to send any message format to the default channel. This is especially useful when your devices publish to fixed topics or when you want to send data to the default channel from devices that can't process the data into the connector's supported message format (p. 304).

You define the topic syntax in the subscription that you create to connect this connector to the data source. We recommend that you specify a topic path instead of using only the `#` wildcard in your subscriptions.

The connector data doesn't parse the messages that are published to this input topic. All input messages are treated as binary data. Before sending the messages to AWS IoT Analytics, the connector encodes and formats them to conform with `BatchPutMessage` API requirements:

- The connector base64-encodes the raw data and includes the encoded payload in an outbound `BatchPutMessage` request.
- The connector generates and assigns an ID to each input message.

**Note**
The connector's response output doesn't include an ID correlation for these input messages.

**Message properties**
None.

**Output Data**

This connector publishes status information as output data. This information contains the response returned by AWS IoT Analytics for each input message that it receives and sends to AWS IoT Analytics.

**Topic filter**

`iotanalytics/messages/put/status`

**Example output: Success**

```json
{
    "response" : {
        "status" : "success"
    },
    "id" : "req123"
}
```
Example output: Failure

```json
{
  "response": {
    "status": "fail",
    "error": "ResourceNotFoundException",
    "error_message": "A resource with the specified name could not be found."
  },
  "id": "req123"
}
```

Note
If the connector detects a retryable error (for example, connection errors), it retries the publish in the next batch. Exponential backoff is handled by the AWS SDK. Requests with retryable errors are added back to the channel queue for further publishing according to the `IotAnalyticsQueueDropBehavior` parameter.

Limits
This connector is subject to the following limits.

- All limits imposed by the AWS SDK for Python (boto3) for the AWS IoT Analytics `batch_put_message` action.
- All limits imposed by the AWS IoT Analytics `BatchPutMessage` API. For more information, see `AWS IoT Analytics Limits` in the `AWS General Reference`.
  - 100,000 messages per second per channel.
  - 100 messages per batch.
  - 128 KB per message.

This API uses channel names (not channel ARNs), so sending data to cross-region or cross-account channels is not supported.

- All limits imposed by the AWS IoT Greengrass Core. For more information, see `AWS IoT Greengrass Core Limits` in the `AWS General Reference`.

The following limits might be especially applicable:

- Maximum size of messages sent by a device is 128 KB.
- Maximum message queue size in the Greengrass core router is 2.5 MB.
- Maximum length of a topic string is 256 bytes of UTF-8 encoded characters.

- This connector can be used only in AWS Regions that are supported by both AWS IoT Greengrass and AWS IoT Analytics. Currently, this includes the following Regions:
  - US East (Ohio) - us-east-2
  - US East (N. Virginia) - us-east-1
  - US West (Oregon) - us-west-2
  - Asia Pacific (Tokyo) - ap-northeast-1
  - EU (Frankfurt) - eu-central-1
  - EU (Ireland) - eu-west-1

Licenses

The IoT Analytics connector includes the following third-party software/licensing:

- AWS SDK for Python (Boto 3)/Apache 2.0
This connector is released under the Greengrass Core Software License Agreement.

**Changelog**

The following table describes the changes in each version of the connector.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Fix to reduce excessive logging.</td>
</tr>
<tr>
<td>1</td>
<td>Initial release.</td>
</tr>
</tbody>
</table>

A Greengrass group can contain only one version of the connector at a time.

**See Also**

- Integrate with Services and Protocols Using Connectors (p. 283)
- the section called “Get Started with Connectors (Console)” (p. 403)
- the section called “Get Started with Connectors (CLI)” (p. 413)

**Kinesis Firehose**

The Kinesis Firehose connector (p. 283) publishes data through an Amazon Kinesis Data Firehose delivery stream to destinations such as Amazon S3, Amazon Redshift, or Amazon Elasticsearch Service.

This connector is a data producer for a Kinesis delivery stream. It receives input data on an MQTT topic, and sends the data to a specified delivery stream. The delivery stream then sends the data record to the configured destination (for example, an S3 bucket).

This connector has the following versions.

<table>
<thead>
<tr>
<th>Version</th>
<th>ARN</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>arn:aws:greengrass:region::/connectors/KinesisFirehose/versions/3</td>
</tr>
<tr>
<td>2</td>
<td>arn:aws:greengrass:region::/connectors/KinesisFirehose/versions/2</td>
</tr>
<tr>
<td>1</td>
<td>arn:aws:greengrass:region::/connectors/KinesisFirehose/versions/1</td>
</tr>
</tbody>
</table>

For information about version changes, see the Changelog (p. 315).

**Requirements**

This connector has the following requirements:

Versions 2 and 3

- AWS IoT Greengrass Core Software v1.7 or later.
• **Python** version 2.7 installed on the core device and added to the PATH environment variable.

• A configured Kinesis delivery stream. For more information, see *Creating an Amazon Kinesis Data Firehose Delivery Stream* in the *Amazon Kinesis Firehose Developer Guide*.

• An IAM policy added to the Greengrass group role that allows the `firehose:PutRecord` and `firehose:PutRecordBatch` actions on the target delivery stream, as shown in the following example:

```json
{
    "Version":"2012-10-17",
    "Statement": [ 
        {
            "Sid":"Stmt1528133056761",
            "Action": [ 
                "firehose:PutRecord",
                "firehose:PutRecordBatch"
            ],
            "Effect": "Allow",
            "Resource": [ 
                "arn:aws:firehose:region:account-id:deliverystream/stream-name"
            ]
        }
    ]
}
```

This connector allows you to dynamically override the default delivery stream in the input message payload. If your implementation uses this feature, the IAM policy should include all target streams as resources. 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*.

**Version 1**

• AWS IoT Greengrass Core Software v1.7 or later.

• **Python** version 2.7 installed on the core device and added to the PATH environment variable.

• A configured Kinesis delivery stream. For more information, see *Creating an Amazon Kinesis Data Firehose Delivery Stream* in the *Amazon Kinesis Firehose Developer Guide*.

• An IAM policy added to the Greengrass group role that allows the `firehose:PutRecord` action on the target delivery stream, as shown in the following example:

```json
{
    "Version":"2012-10-17",
    "Statement": [ 
        {
            "Sid":"Stmt1528133056761",
            "Action": [ 
                "firehose:PutRecord"
            ],
            "Effect": "Allow",
            "Resource": [ 
                "arn:aws:firehose:region:account-id:deliverystream/stream-name"
            ]
        }
    ]
}
```

This connector allows you to dynamically override the default delivery stream in the input message payload. If your implementation uses this feature, the IAM policy should include all target streams as resources. 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.

Connector Parameters
This connector provides the following parameters:

Versions 2 and 3

DefaultDeliveryStreamArn
The ARN of the default Kinesis Data Firehose delivery stream to send data to. The destination stream can be overridden by the delivery_stream_arn property in the input message payload.

Note
The group role must allow the appropriate actions on all target delivery streams. For more information, see the section called "Requirements" (p. 307).

Display name in console: Default delivery stream ARN
Required: true
Type: string
Valid pattern: arn:aws:firehose:([-a-z]+-[a-z]+-\d{1}):
\d{12}:deliverystream/([-a-zA-Z0-9_\-\s.]+)$

DeliveryStreamQueueSize
The maximum number of records to retain in memory before new records for the same delivery stream are rejected. The minimum value is 2000.

Display name in console: Maximum number of records to buffer (per stream)
Required: true
Type: string
Valid pattern: ^([2-9]\d{3}|[1-9]\d{4,})$

MemorySize
The amount of memory (in KB) to allocate to this connector.

Display name in console: Memory size
Required: true
Type: string
Valid pattern: ^[0-9]+$

PublishInterval
The interval (in seconds) for publishing records to Kinesis Data Firehose. To disable batching, set this value to 0.

Display name in console: Publish interval
Required: true
Type: string
Valid values: 0 - 900
Valid pattern: \[0-9][1-9]\d|[1-9]\d|900

Version 1

**DefaultDeliveryStreamArn**

The ARN of the default Kinesis Data Firehose delivery stream to send data to. The destination stream can be overridden by the `delivery_stream_arn` property in the input message payload.

*Note*

The group role must allow the appropriate actions on all target delivery streams. For more information, see the section called “Requirements” (p. 307).

Display name in console: Default delivery stream ARN

Required: true

Type: string

Valid pattern: `arn:aws:firehose:(\[a-z\]{2}-\[a-z]+-\d{1}):\(\d{12}\):deliverystream/(\[a-zA-Z0-9_\-\.]\+)+`$`

**Example**

**Create Connector Example (CLI)**

The following CLI command creates a `ConnectorDefinition` with an initial version that contains the connector.

```
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-version '{
  "Connectors": [
    {
      "Id": "MyKinesisFirehoseConnector",
      "ConnectorArn": "arn:aws:greengrass:region::/connectors/KinesisFirehose/versions/3",
      "Parameters": {
        "DeliveryStreamQueueSize": "5000",
        "MemorySize": "65535",
        "PublishInterval": "10"
      }
    }
  ]
}
```

In the AWS IoT Greengrass console, you can add a connector from the group's Connectors page. For more information, see the section called “Get Started with Connectors (Console)” (p. 403).

**Input Data**

This connector accepts stream content on MQTT topics, and then sends the content to the target delivery stream. It accepts two types of input data:

- JSON data on the `kinesisfirehose/message` topic.
• Binary data on the kinesisfirehose/message(binary/# topic.

Versions 2 and 3

**Topic filter:** kinesisfirehose/message

Use this topic to send a message that contains JSON data.

**Message properties**

**request**

The data to send to the delivery stream and the target delivery stream, if different from the default stream.

Required: true
Type: object that includes the following properties:

**data**

The data to send to the delivery stream.

Required: true
Type: bytes

**delivery_stream_arn**

The ARN of the target Kinesis delivery stream. Include this property to override the default delivery stream.

Required: false
Type: string
Valid pattern: arn:aws:firehose:([a-z]{2}-[a-z]+-[0-9]{1}):(\d{12}):deliverystream/([a-zA-Z0-9-_\.]+)$

**id**

An arbitrary ID for the request. This property is used to map an input request to an output response. When specified, the id property in the response object is set to this value. If you don’t use this feature, you can omit this property or specify an empty string.

Required: false
Type: string
Valid pattern: .*

**Example input**

```json
{
   "request": {
      "data": "Data to send to the delivery stream."
   },
   "id": "request123"
}
```
**Topic filter:** kinesisfirehose/message/binary/#

Use this topic to send a message that contains binary data. The connector doesn’t parse binary data. The data is streamed 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.

**Version 1**

**Topic filter:** kinesisfirehose/message

Use this topic to send a message that contains JSON data.

**Message properties**

- **request**
  
  The data to send to the delivery stream and the target delivery stream, if different from the default stream.

  Required: true

  Type: object that includes the following properties:

  - **data**
    
    The data to send to the delivery stream.

    Required: true

    Type: bytes

  - **delivery_stream_arn**
    
    The ARN of the target Kinesis delivery stream. Include this property to override the default delivery stream.

    Required: false

    Type: string

    Valid pattern: 
    
    `arn:aws:firehose:([a-z]{2}-[a-z]+-\d{1}):(\d{12}):deliverystream/([a-zA-Z0-9\-_\.]+)\+$`

  - **id**
    
    An arbitrary ID for the request. This property is used to map an input request to an output response. When specified, the id property in the response object is set to this value. If you don’t use this feature, you can omit this property or specify an empty string.

    Required: false

    Type: string

    Valid pattern: .*

**Example input**

```json
{}
```
"request": {  
      "data": "Data to send to the delivery stream."  
},  
  "id": "request123"  
}

**Topic filter**: kinesisfirehose/message/binary/#

Use this topic to send a message that contains binary data. The connector doesn't parse binary data. The data is streamed 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 connector publishes status information as output data.

**Versions 2 and 3**

**Topic filter**

kinesisfirehose/message/status

**Example output**

The response contains the status of each data record sent in the batch.

```json
{
  "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 connector detects a retryable error (for example, connection errors), it retries the publish in the next batch. Exponential backoff is handled by the AWS SDK. Requests that fail with retryable errors are added back to the end of the queue for further publishing.
Version 1

**Topic filter**

```
kinesisfirehose/message/status
```

Example output: Success

```
{
   "response": {
      "firehose_record_id": "llxfuuuFomkpJYzt/34ZU/r8JYPf8Wyf7AXqlXm",
      "status": "success"
   },
   "id": "request123"
}
```

Example output: Failure

```
{
   "response": {
      "error": "ResourceNotFoundException",
      "error_message": "An error occurred (ResourceNotFoundException) when calling the PutRecord operation: Firehose test1 not found under account 123456789012.",
      "status": "fail"
   },
   "id": "request123"
}
```

**Usage Example**

The following example Lambda function sends an input message to the connector. This message contains JSON data.

**Note**
This Python function uses the [AWS IoT Greengrass Core SDK (p. 175)](https://docs.aws.amazon.com/greengrass/latest/developerguide/gsg-pip-install.html) to publish an MQTT message. You can use the following `pip` command to install the Python version of the SDK on your core device:

```
pip install greengrasssdk
```

```python
import greengrasssdk
import json

iot_client = greengrasssdk.client('iot-data')
send_topic = 'kinesisfirehose/message'

def create_request_with_all_fields():
    return {
        "request": {
            "data": "Message from Firehose Connector Test"
        },
        "id": "req_123"
    }

def publish_basic_message():
    messageToPublish = create_request_with_all_fields()
    print "Message To Publish: ", messageToPublish
    iot_client.publish(topic=send_topic,
                       payload=json.dumps(messageToPublish))
```
publish_basic_message()

def function_handler(event, context):
    return

Licenses

The Kinesis Firehose connector includes the following third-party software/licensing:

- AWS SDK for Python (Boto 3)/Apache 2.0

This connector is released under the Greengrass Core Software License Agreement.

Changelog

The following table describes the changes in each version of the connector.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Fix to reduce excessive logging and other minor bug fixes.</td>
</tr>
<tr>
<td>2</td>
<td>Added support for sending batched data records to Kinesis Data Firehose at a specified interval.</td>
</tr>
<tr>
<td></td>
<td>• Also requires the firehose:PutRecordBatch action in the group role.</td>
</tr>
<tr>
<td></td>
<td>• New MemorySize, DeliveryStreamQueueSize, and PublishInterval parameters.</td>
</tr>
<tr>
<td></td>
<td>• Output message contains an array of status responses for the published data records.</td>
</tr>
<tr>
<td>1</td>
<td>Initial release.</td>
</tr>
</tbody>
</table>

A Greengrass group can contain only one version of the connector at a time.

See Also

- Integrate with Services and Protocols Using Connectors (p. 283)
- the section called “Get Started with Connectors (Console)” (p. 403)
- the section called “Get Started with Connectors (CLI)” (p. 413)
- What Is Amazon Kinesis Data Firehose? in the Amazon Kinesis Developer Guide

ML Feedback Connector

The ML Feedback connector makes it easier to access your machine learning (ML) model data for model retraining and analysis. The connector:

- Uploads input data (samples) used by your ML model to Amazon S3. Model input can be in any format, such as images, JSON, or audio. After samples are uploaded to the cloud, you can use them to retrain
the model to improve the accuracy and precision of its predictions. For example, you can use Amazon SageMaker Ground Truth to label your samples and Amazon SageMaker to retrain the model.

- Publishes the prediction results from the model as MQTT messages. This lets you monitor and analyze the inference quality of your model in real time. You can also store prediction results and use them to analyze trends over time.
- Publishes metrics about sample uploads and sample data to Amazon CloudWatch.

To configure this connector, you describe your supported feedback configurations in JSON format. A feedback configuration defines properties such as the destination Amazon S3 bucket, content type, and sampling strategy (p. 320). (A sampling strategy is used to determine which samples to upload.)

You can use the ML Feedback connector in the following scenarios:

- With user-defined Lambda functions. Your local inference Lambda functions use the AWS IoT Greengrass Machine Learning SDK to invoke this connector and pass in the target feedback configuration, model input, and model output (prediction results). For an example, see the section called “Usage Example” (p. 325).
- With the ML Image Classification connector (p. 326) (v2). To use this connector with the ML Image Classification connector, configure the MLFeedbackConnectorConfigId parameter for the ML Image Classification connector.
- With the ML Object Detection connector (p. 343). To use this connector with the ML Object Detection connector, configure the MLFeedbackConnectorConfigId parameter for the ML Object Detection connector.

**ARN:** arn:aws:greengrass:region::/connectors/MLFeedback/versions/1

**Requirements**

This connector has the following requirements:

- AWS IoT Greengrass Core Software v1.9.3 or later.
- Python version 3.7 installed on the core device and added to the PATH environment variable.
- One or more Amazon S3 buckets. The number of buckets you use depends on your sampling strategy.
- An IAM policy added to the Greengrass group role that allows the s3:PutObject action on objects in the destination Amazon S3 bucket, as shown in the following example.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "s3:PutObject",
      "Resource": [
        "arn:aws:s3:::bucket-name/**"
      ]
    }
  ]
}
```

The policy should include all destination buckets as resources. 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.

- The CloudWatch Metrics connector (p. 289) added to the Greengrass group and configured. This is required only if you want to use the metrics reporting feature.
- AWS IoT Greengrass Machine Learning SDK (p. 176) v1.1.0 is required to interact with this connector.
Parameters

FeedbackConfigurationMap

A set of one or more feedback configurations that the connector can use to upload samples to Amazon S3. A feedback configuration defines parameters such as the destination bucket, content type, and sampling strategy (p. 320). When this connector is invoked, the calling Lambda function or connector specifies a target feedback configuration.

Display name in console: Feedback configuration map
Required: true
Type: A well-formed JSON string that defines the set of supported feedback configurations. For an example, see the section called “FeedbackConfigurationMap Example” (p. 319).

The ID of a feedback configuration object has the following requirements.

The ID:
- Must be unique across configuration objects.
- Must begin with a letter or number. Can contain lowercase and uppercase letters, numbers, and hyphens.
- Must be 2 - 63 characters in length.

Required: true
Type: string
Valid pattern: ^[a-zA-Z0-9][a-zA-Z0-9-]{1,62}$
Examples: MyConfig0, config-a, 12id

The body of a feedback configuration object contains the following properties.

s3-bucket-name
The name of the destination Amazon S3 bucket.

Note
The group role must allow the s3:PutObject action on all destination buckets. For more information, see the section called “Requirements” (p. 316).

Required: true
Type: string
Valid pattern: ^[a-zA-Z0-9\./-]{3,63}$

content-type
The content type of the samples to upload. All content for an individual feedback configuration must be of the same type.

Required: true
Type: string
Examples: image/jpeg, application/json, audio/ogg
s3-prefix

The key prefix to use for uploaded samples. A prefix is similar to a directory name. It allows you to store similar data under the same directory in a bucket. For more information, see Object Key and Metadata in the Amazon Simple Storage Service Developer Guide.

Required: false
Type: string

file-ext

The file extension to use for uploaded samples. Must be a valid file extension for the content type.

Required: false
Type: string

Examples: jpg, json, ogg

sampling-strategy

The sampling strategy (p. 320) to use to filter which samples to upload. If omitted, the connector tries to upload all the samples that it receives.

Required: false
Type: A well-formed JSON string that contains the following properties.

strategy-name

The name of the sampling strategy.

Required: true
Type: string

Valid values: RANDOM_SAMPLING, LEAST_CONFIDENCE, MARGIN, or ENTROPY

rate

The rate for the Random (p. 320) sampling strategy.

Required: true if strategy-name is RANDOM_SAMPLING.
Type: number

Valid values: 0.0 - 1.0

threshold

The threshold for the Least Confidence (p. 320), Margin (p. 320), or Entropy (p. 321) sampling strategy.

Required: true if strategy-name is LEAST_CONFIDENCE, MARGIN, or ENTROPY.
Type: number

Valid values:
• 0.0 - 1.0 for the LEAST_CONFIDENCE or MARGIN strategy.
• 0.0 - no limit for the ENTROPY strategy.

RequestLimit

The maximum number of requests that the connector can process at a time.
You can use this parameter to restrict memory consumption by limiting the number of requests that the connector processes at the same time. Requests that exceed this limit are ignored.

Display name in console: **Request limit**

- **Required:** false
- **Type:** string
- **Valid values:** 0 - 999
- **Valid pattern:** ^$|^[0-9]{1,3}$

**Create Connector Example (CLI)**

The following CLI command creates a `ConnectorDefinition` with an initial version that contains the ML Feedback connector.

```bash
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-version '{
  "Connectors": [
    {
      "Id": "MyMLFeedbackConnector",
      "ConnectorArn": "arn:aws:greengrass:region::/connectors/MLFeedback/versions/1",
      "Parameters": {
        "FeedbackConfigurationMap": "{ "RandomSamplingConfiguration": { "s3-bucket-name": "my-aws-bucket-random-sampling", "content-type": "image/png", "file-ext": "png", "sampling-strategy": { "strategy-name": "RANDOM_SAMPLING", "rate": 0.5 } }, "LeastConfidenceConfiguration": { "s3-bucket-name": "my-aws-bucket-least-confidence-sampling", "content-type": "image/png", "file-ext": "png", "sampling-strategy": { "strategy-name": "LEAST_CONFIDENCE", "threshold": 0.4 } } },
        "RequestLimit": "10"
      }
    }
  ]
}
```

**FeedbackConfigurationMap Example**

The following is an expanded example value for the `FeedbackConfigurationMap` parameter. This example includes several feedback configurations that use different sampling strategies.

```json
{
  "ConfigID1": {
    "s3-bucket-name": "my-aws-bucket-random-sampling",
    "content-type": "image/png",
    "file-ext": "png",
    "sampling-strategy": {
      "strategy-name": "RANDOM_SAMPLING",
      "rate": 0.5
    }
  },
  "ConfigID2": {
    "s3-bucket-name": "my-aws-bucket-margin-sampling",
    "content-type": "image/png",
    "file-ext": "png",
    "sampling-strategy": {
      "strategy-name": "MARGIN",
      "threshold": 0.4
    }
  }
}
```
"ConfigID3": {
  "s3-bucket-name": "my-aws-bucket-least-confidence-sampling",
  "content-type": "image/png",
  "file-ext": "png",
  "sampling-strategy": {
    "strategy-name": "LEAST_CONFIDENCE",
    "threshold": 0.4
  }
},

"ConfigID4": {
  "s3-bucket-name": "my-aws-bucket-entropy-sampling",
  "content-type": "image/png",
  "file-ext": "png",
  "sampling-strategy": {
    "strategy-name": "ENTROPY",
    "threshold": 2
  }
},

"ConfigID5": {
  "s3-bucket-name": "my-aws-bucket-no-sampling",
  "s3-prefix": "DeviceA",
  "content-type": "application/json"
}

**Sampling Strategies**

The connector supports four sampling strategies that determine whether to upload samples that are passed to the connector. Samples are discrete instances of data that a model uses for a prediction. You can use sampling strategies to filter for the samples that are most likely to improve model accuracy.

**RANDOM_SAMPLING**

Randomly uploads samples based on the supplied rate. It uploads a sample if a randomly generated value is less than the rate. The higher the rate, the more samples are uploaded.

*Note*  
This strategy disregards any model prediction that is supplied.

**LEAST_CONFIDENCE**

Uploads samples whose maximum confidence probability falls below the supplied threshold.  

**Example scenario:**

Threshold: .6  
Model prediction: [.2, .2, .4, .2]  
Maximum confidence probability: .4  

**Result:**

Use the sample because maximum confidence probability (.4) <= threshold (.6).

**MARGIN**

Uploads samples if the margin between the top two confidence probabilities falls within the supplied threshold. The margin is the difference between the top two probabilities.

**Example scenario:**

Threshold: .02  
Model prediction: [.3, .35, .34, .01]  
Top two confidence probabilities: [.35, .34]
Margin: \(0.01 (0.35 - 0.34)\)

Result:

Use the sample because margin \(0.01 \leq\) threshold \(0.02\).

**ENTROPY**

Uploads samples whose entropy is greater than the supplied threshold. Uses the model prediction's normalized entropy.

Example scenario:

Threshold: \(0.75\)

Model prediction: \([0.5, 0.25, 0.25]\)

Entropy for prediction: \(1.03972\)

Result:

Use sample because entropy \(1.03972\) > threshold \(0.75\).

**Input Data**

User-defined Lambda functions use the `publish` function of the feedback client in the AWS IoT Greengrass Machine Learning SDK to invoke the connector. For an example, see the section called "Usage Example" (p. 325).

**Note**

This connector doesn't accept MQTT messages as input data.

The `publish` function takes the following arguments:

**ConfigId**

The ID of the target feedback configuration. This must match the ID of a feedback configuration defined in the `FeedbackConfigurationMap` (p. 317) parameter for the ML Feedback connector.

Required: true

Type: string

**ModelInput**

The input data that was passed to a model for inference. This input data is uploaded using the target configuration unless it is filtered out based on the sampling strategy.

Required: true

Type: bytes

**ModelPrediction**

The prediction results from the model. The result type can be a dictionary or a list. For example, the prediction results from the ML Image Classification connector is a list of probabilities (such as \([0.25, 0.60, 0.15]\)). This data is published to the `/feedback/message/prediction` topic.

Required: true

Type: dictionary or list of `float` values

**Metadata**

Customer-defined, application-specific metadata that is attached to the uploaded sample and published to the `/feedback/message/prediction` topic. The connector also inserts a `publish-ts` key with a timestamp value into the metadata.
Output Data

This connector publishes data to three MQTT topics:

- Status information from the connector on the feedback/message/status topic.
- Prediction results on the feedback/message/prediction topic.
- Metrics destined for CloudWatch on the cloudwatch/metric/put topic.

You must configure subscriptions to allow the connector to communicate on MQTT topics. For more information, see the section called “Inputs and Outputs” (p. 287).

**Topic filter:** feedback/message/status

Use this topic to monitor the status of sample uploads and dropped samples. The connector publishes to this topic every time that it receives a request.

**Example output:** Sample upload succeeded

```
{ 
    "response": { 
        "status": "success", 
        "s3_response": { 
            "ResponseMetadata": { 
                "HostId": "IOWQ4fDNEXAMPLEQM+ey7N9WgVhSnQ6JEXAMPLEZb7hSQDAK
+Jd1vEXAMPLEa3Km", 
                "RetryAttempts": 1, 
                "HTTPStatusCode": 200, 
                "RequestId": "79104EXAMPLEB723", 
                "HTTPHeaders": { 
                    "content-length": "0", 
                    "x-amz-id-2": "lbbqaDVFQhMlyU3gRvAX1ZIdg8P0WgkCSSFsYFvSwLZk3j7QZhG5EXAMPLEd4/pEXAMPLEuU=" 
                }, 
                "bucket": "greengrass-feedback-connector-data-us-west-2", 
                "ETag": "Yb9c4f172e64458a5f6d74EXAMPLE5628\", 
                "Expiration": " expiry-date="Wed, 17 Jul 2019 00:00:00 GMT\", rule-
id="OGZjYWY3OTgtYW12Zi00ZDllLWE4YmQtNzMyZzEXAMPLEUw\", 
                "x-amz-request-id": "79104EXAMPLEB723", 
                "key": "s3-key-prefix/UUID.file_ext", 
                "ServerSideEncryption": "AES256" 
            }, 
            "id": "5aaa913f-97a3-48ac-5907-18cd96b89eeb"
        }
    }
}
```

The connector adds the bucket and key fields to the response from Amazon S3. For more information about the Amazon S3 response, see PUT Object in the Amazon Simple Storage Service API Reference.
Example output: Sample dropped because of the sampling strategy

```
{  
  "response": {  
    "status": "sample_dropped_by_strategy"  
  },  
  "id": "4bf5aeb0-d1e4-4362-5bb4-87c05de78ba3"
}
```

Example output: Sample upload failed

A failure status includes the error message as the `error_message` value and the exception class as the `error` value.

```
{  
  "response": {  
    "status": "fail",  
    "error_message": "[RequestId: 4bf5aeb0-d1e4-4362-5bb4-87c05de78ba3] Failed to upload model input data due to exception. Model prediction will not be published. Exception type: NoSuchBucket, error: An error occurred (NoSuchBucket) when calling the PutObject operation: The specified bucket does not exist",
    "error": "NoSuchBucket"
  },  
  "id": "4bf5aeb0-d1e4-4362-5bb4-87c05de78ba3"
}
```

Example output: Request throttled because of the request limit

```
{  
  "response": {  
    "status": "fail",  
    "error_message": "Request limit has been reached (max request: 10 ). Dropping request.",  
    "error": "Queue.Full"
  },  
  "id": "4bf5aeb0-d1e4-4362-5bb4-87c05de78ba3"
}
```

**Topic filter:** feedback/message/prediction

Use this topic to listen for predictions based on uploaded sample data. This lets you analyze your model performance in real time. Model predictions are published to this topic only if data is successfully uploaded to Amazon S3. Messages published on this topic are in JSON format. They contain the link to the uploaded data object, the model's prediction, and the metadata included in the request.

You can also store prediction results and use them to report and analyze trends over time. Trends can provide valuable insights. For example, a *decreasing accuracy over time* trend can help you to decide whether the model needs to be retrained.

**Example output**

```
{  
  "source-ref": "s3://greengrass-feedback-connector-data-us-west-2/s3-key-prefix/UUID.file_ext",  
  "model-prediction": [  
    0.5,  
    0.2,  
    0.2,  
    0.1
  ]
}
```
Tip
You can configure the IoT Analytics connector (p. 299) to subscribe to this topic and send the information to AWS IoT Analytics for further or historical analysis.

Topic filter: cloudwatch/metric/put

This is the output topic used to publish metrics to CloudWatch. This feature requires that you install and configure the CloudWatch Metrics connector (p. 289).

Metrics include:
- The number of uploaded samples.
- The size of uploaded samples.
- The number of errors from uploads to Amazon S3.
- The number of dropped samples based on the sampling strategy.
- The number of throttled requests.

Example output: Size of the data sample (published before the actual upload)

```
{
  "request": {
    "namespace": "GreengrassFeedbackConnector",
    "metricData": {
      "value": 47592,
      "unit": "Bytes",
      "metricName": "SampleSize"
    }
  }
}
```

Example output: Sample upload succeeded

```
{
  "request": {
    "namespace": "GreengrassFeedbackConnector",
    "metricData": {
      "value": 1,
      "unit": "Count",
      "metricName": "SampleUploadSuccess"
    }
  }
}
```

Example output: Sample upload succeeded and prediction result published

```
{
  "request": {
    "namespace": "GreengrassFeedbackConnector",
    "metricData": {
      "value": 1,
      "unit": "Count",
      "metricName": "SampleAndPredictionPublished"
    }
  }
}
```
Example output: Sample upload failed

```json
{
    "request": {
        "namespace": "GreengrassFeedbackConnector",
        "metricData": {
            "value": 1,
            "unit": "Count",
            "metricName": "SampleUploadFailure"
        }
    }
}
```

Example output: Sample dropped because of the sampling strategy

```json
{
    "request": {
        "namespace": "GreengrassFeedbackConnector",
        "metricData": {
            "value": 1,
            "unit": "Count",
            "metricName": "SampleNotUsed"
        }
    }
}
```

Example output: Request throttled because of the request limit

```json
{
    "request": {
        "namespace": "GreengrassFeedbackConnector",
        "metricData": {
            "value": 1,
            "unit": "Count",
            "metricName": "ErrorRequestThrottled"
        }
    }
}
```

Usage Example

The following example is a user-defined Lambda function that uses the AWS IoT Greengrass Machine Learning SDK (p. 176) to send data to the ML Feedback connector.

**Note**

You can download the AWS IoT Greengrass Machine Learning SDK from the AWS IoT Greengrass downloads page (p. 23).

```python
import json
import logging
import os
import sys
import greengrass_machine_learning_sdk as ml

client = ml.client('feedback')

try:
    feedback_config_id = os.environ['FEEDBACK_CONFIG_ID']
```
model_input_data_dir = os.environ["MODEL_INPUT_DIR"]
model_prediction_str = os.environ["MODEL_PREDICTIONS"]
model_prediction = json.loads(model_prediction_str)
except Exception as e:
    logging.info("Failed to open environment variables. Failed with exception:
    
    sys.exit(1)

try:
    with open(os.path.join(model_input_data_dir, os.listdir(model_input_data_dir)[0]), 'rb') as f:
        content = f.read()
except Exception as e:
    logging.info("Failed to open model input directory. Failed with exception:
    
    sys.exit(1)

def invoke_feedback_connector():
    logging.info("Invoking feedback connector.")
    try:
        client.publish(
            ConfigId=feedback_config_id,
            ModelInput=content,
            ModelPrediction=model_prediction
        )
    except Exception as e:
        logging.info("Exception raised when invoking feedback connector:
        
        sys.exit(1)

invoke_feedback_connector()

def function_handler(event, context):
    return

Licenses

The ML Feedback connector includes the following third-party software/licensing:

- AWS SDK for Python (Boto 3)/Apache 2.0
- six/MIT

This connector is released under the Greengrass Core Software License Agreement.

See Also

- Integrate with Services and Protocols Using Connectors (p. 283)
- the section called “Get Started with Connectors (Console)” (p. 403)
- the section called “Get Started with Connectors (CLI)” (p. 413)

ML Image Classification Connector

The ML Image Classification connectors (p. 283) provide a machine learning (ML) inference service that runs on the AWS IoT Greengrass core. This local inference service performs image classification using a model trained by the Amazon SageMaker image classification algorithm.

User-defined Lambda functions use the AWS IoT Greengrass Machine Learning SDK to submit inference requests to the local inference service. The service runs inference locally and returns probabilities that the input image belongs to specific categories.
AWS IoT Greengrass provides the following versions of this connector, which is available for multiple platforms.

**Version 2**

<table>
<thead>
<tr>
<th>Connector</th>
<th>Description and ARN</th>
</tr>
</thead>
<tbody>
<tr>
<td>ML Image Classification Aarch64 JTX2</td>
<td>Image classification inference service for NVIDIA Jetson TX2. Supports GPU acceleration. ARN: <code>arn:aws:greengrass:region::/connectors/ImageClassificationAarch64JTX2/versions/2</code></td>
</tr>
<tr>
<td>ML Image Classification x86_64</td>
<td>Image classification inference service for x86_64 platforms. ARN: <code>arn:aws:greengrass:region::/connectors/ImageClassificationx86-64/versions/2</code></td>
</tr>
<tr>
<td>ML Image Classification ARMv7</td>
<td>Image classification inference service for ARMv7 platforms. ARN: <code>arn:aws:greengrass:region::/connectors/ImageClassificationARMv7/versions/2</code></td>
</tr>
</tbody>
</table>

**Version 1**

<table>
<thead>
<tr>
<th>Connector</th>
<th>Description and ARN</th>
</tr>
</thead>
<tbody>
<tr>
<td>ML Image Classification Aarch64 JTX2</td>
<td>Image classification inference service for NVIDIA Jetson TX2. Supports GPU acceleration. ARN: <code>arn:aws:greengrass:region::/connectors/ImageClassificationAarch64JTX2/versions/1</code></td>
</tr>
<tr>
<td>ML Image Classification x86_64</td>
<td>Image classification inference service for x86_64 platforms. ARN: <code>arn:aws:greengrass:region::/connectors/ImageClassificationx86-64/versions/1</code></td>
</tr>
<tr>
<td>ML Image Classification ARMv7</td>
<td>Image classification inference service for ARMv7 platforms. ARN: <code>arn:aws:greengrass:region::/connectors/ImageClassificationARMv7/versions/1</code></td>
</tr>
</tbody>
</table>

For information about version changes, see the [Changelog (p. 342)](https://aws.amazon.com/).
Requirements

These connectors have the following requirements:

Version 2

- AWS IoT Greengrass Core Software v1.9.3 or later.
- Python version 3.7 installed on the core device and added to the PATH environment variable.
- Dependencies for the Apache MXNet framework installed on the core device. For more information, see the section called “Installing MXNet Dependencies” (p. 336).
- An ML resource (p. 221) in the Greengrass group that references an Amazon SageMaker model source. This model must be trained by the Amazon SageMaker image classification algorithm. For more information, see Image Classification Algorithm in the Amazon SageMaker Developer Guide.
- The ML Feedback connector (p. 315) added to the Greengrass group and configured. This is required only if you want to use the connector to upload model input data and publish predictions to an MQTT topic.
- An IAM policy added to the Greengrass group role that allows the `sagemaker:DescribeTrainingJob` action on the target training job, as shown in the following example.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "sagemaker:DescribeTrainingJob"
            ],
        }
    ]
}
```

You can grant granular or conditional access to resources (for example, by using a wildcard * naming scheme). If you change the target training job in the future, make sure to update the group role. For more information, see Adding and Removing IAM Policies in the IAM User Guide.

- AWS IoT Greengrass Machine Learning SDK (p. 176) v1.1.0 is required to interact with this connector.

Version 1

- AWS IoT Greengrass Core Software v1.7 or later.
- Python version 2.7 installed on the core device and added to the PATH environment variable.
- Dependencies for the Apache MXNet framework installed on the core device. For more information, see the section called “Installing MXNet Dependencies” (p. 336).
- An ML resource (p. 221) in the Greengrass group that references an Amazon SageMaker model source. This model must be trained by the Amazon SageMaker image classification algorithm. For more information, see Image Classification Algorithm in the Amazon SageMaker Developer Guide.
- An IAM policy added to the Greengrass group role that allows the `sagemaker:DescribeTrainingJob` action on the target training job, as shown in the following example.

```
{
    "Version": "2012-10-17",
}
```
"Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sagemaker:DescribeTrainingJob"
      ],
    }
  ]
}

You can grant granular or conditional access to resources (for example, by using a wildcard * naming scheme). If you change the target training job in the future, make sure to update the group role. For more information, see Adding and Removing IAM Policies in the IAM User Guide.

- AWS IoT Greengrass Machine Learning SDK (p. 176) v1.0.0 or later is required to interact with this connector.

## Connector Parameters

These connectors provide the following parameters.

### Version 2

**MLModelDestinationPath**

The absolute local path of the ML resource inside the Lambda environment. This is the destination path that's specified for the ML resource.

**Note**

- If you created the ML resource in the console, this is the local path.

Display name in console: **Model destination path**

Required: true

Type: string

Valid pattern: .+

**MLModelResourceId**

The ID of the ML resource that references the source model.

Display name in console: **SageMaker job ARN resource**

Required: true

Type: string

Valid pattern: [a-zA-Z0-9:-_]+

**MLModelSageMakerJobArn**

The ARN of the Amazon SageMaker training job that represents the Amazon SageMaker model source. The model must be trained by the Amazon SageMaker image classification algorithm.

Display name in console: **SageMaker job ARN**

Required: true
Type: string


LocalInferenceServiceName

The name for the local inference service. User-defined Lambda functions invoke the service by passing the name to the `invoke_inference_service` function of the AWS IoT Greengrass Machine Learning SDK. For an example, see the section called “Usage Example” (p. 334).

Display name in console: **Local inference service name**

Required: true

Type: string

Valid pattern: [a-zA-Z0-9][a-zA-Z0-9-]{1,62}

LocalInferenceServiceTimeoutSeconds

The amount of time (in seconds) before the inference request is terminated. The minimum value is 1.

Display name in console: **Timeout (second)**

Required: true

Type: string

Valid pattern: [1-9][0-9]*

LocalInferenceServiceMemoryLimitKB

The amount of memory (in KB) that the service has access to. The minimum value is 1.

Display name in console: **Memory limit (KB)**

Required: true

Type: string

Valid pattern: [1-9][0-9]*

GPUAcceleration

The CPU or GPU (accelerated) computing context. This property applies to the ML Image Classification Aarch64 JTX2 connector only.

Display name in console: **GPU acceleration**

Required: true

Type: string

Valid values: CPU or GPU

MLFeedbackConnectorConfigId

The ID of the feedback configuration to use to upload model input data. This must match the ID of a feedback configuration defined for the ML Feedback connector (p. 315).
This parameter is required only if you want to use the ML Feedback connector to upload model input data and publish predictions to an MQTT topic.

Display name in console: **ML Feedback connector configuration ID**

Required: false

Type: string

Valid pattern: `^$|^[a-zA-Z0-9][a-zA-Z0-9-]{1,62}$`

**Version 1**

**MLModelDestinationPath**

The absolute local path of the ML resource inside the Lambda environment. This is the destination path that's specified for the ML resource.

**Note**

If you created the ML resource in the console, this is the local path.

Display name in console: **Model destination path**

Required: true

Type: string

Valid pattern: `.+`

**MLModelResourceId**

The ID of the ML resource that references the source model.

Display name in console: **SageMaker job ARN resource**

Required: true

Type: string

Valid pattern: `[a-zA-Z0-9:._-]*`

**MLModelSageMakerJobArn**

The ARN of the Amazon SageMaker training job that represents the Amazon SageMaker model source. The model must be trained by the Amazon SageMaker image classification algorithm.

Display name in console: **SageMaker job ARN**

Required: true

Type: string


**LocalInferenceServiceName**

The name for the local inference service. User-defined Lambda functions invoke the service by passing the name to the `invoke_inference_service` function of the AWS IoT Greengrass Machine Learning SDK. For an example, see the section called “Usage Example” (p. 334).
Display name in console: **Local inference service name**

Required: true

Type: string

Valid pattern: `[a-zA-Z0-9][a-zA-Z0-9-]{1,62}`

**LocalInferenceServiceTimeoutSeconds**

The amount of time (in seconds) before the inference request is terminated. The minimum value is 1.

Display name in console: **Timeout (second)**

Required: true

Type: string

Valid pattern: `[1-9][0-9]*`

**LocalInferenceServiceMemoryLimitKB**

The amount of memory (in KB) that the service has access to. The minimum value is 1.

Display name in console: **Memory limit (KB)**

Required: true

Type: string

Valid pattern: `[1-9][0-9]*`

**GPUAcceleration**

The CPU or GPU (accelerated) computing context. This property applies to the ML Image Classification Aarch64 JTX2 connector only.

Display name in console: **GPU acceleration**

Required: true

Type: string

Valid values: CPU or GPU

### Create Connector Example (CLI)

The following CLI commands create a `ConnectorDefinition` with an initial version that contains an ML Image Classification connector.

#### Example: CPU Instance

This example creates an instance of the ML Image Classification Armv7l connector.

```bash
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-version '{
  "Connectors": [
    {
      "Id": "MyImageClassificationConnector",
      "Type": "string",
      "Valid pattern": 
```
"ConnectorArn": "arn:aws:greengrass:region::/connectors/ ImageClassificationARMv7/versions/2",
"Parameters": {
    "MLModelDestinationPath": "/path-to-model",
    "MLModelResourceId": "my-ml-resource",
    "LocalInferenceServiceName": "imageClassification",
    "LocalInferenceServiceTimeoutSeconds": "10",
    "LocalInferenceServiceMemoryLimitKB": "500000",
    "MLFeedbackConnectorConfigId": "MyConfig0"
}
}
}
]

Example: GPU Instance

This example creates an instance of the ML Image Classification Aarch64 JTX2 connector, which supports GPU acceleration on an NVIDIA Jetson TX2 board.

```bash
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-version '{
    "Connectors": [
        {
            "Id": "MyImageClassificationConnector",
            "ConnectorArn": "arn:aws:greengrass:region::/connectors/ ImageClassificationAarch64JTX2/versions/2",
            "Parameters": {
                "MLModelDestinationPath": "/path-to-model",
                "MLModelResourceId": "my-ml-resource",
                "LocalInferenceServiceName": "imageClassification",
                "LocalInferenceServiceTimeoutSeconds": "10",
                "LocalInferenceServiceMemoryLimitKB": "500000",
                "GPUAcceleration": "GPU",
                "MLFeedbackConnectorConfigId": "MyConfig0"
            }
        }
    ]
}
'
```

Note
The Lambda function in these connectors have a long-lived (p. 187) lifecycle.

In the AWS IoT Greengrass console, you can add a connector from the group's Connectors page. For more information, see the section called “Get Started with Connectors (Console)” (p. 403).

Input Data

These connectors accept an image file as input. Input image files must be in jpeg or png format. For more information, see the section called "Usage Example" (p. 334).

These connectors don't accept MQTT messages as input data.

Output Data

These connectors return a formatted prediction for the object identified in the input image:
AWS IoT Greengrass Developer Guide
ML Image Classiﬁcation

[0.3,0.1,0.04,...]

The prediction contains a list of values that correspond with the categories used in the training
dataset during model training. Each value represents the probability that the image falls under the
corresponding category. The category with the highest probability is the dominant prediction.
These connectors don't publish MQTT messages as output data.

Usage Example
The following example Lambda function uses the AWS IoT Greengrass Machine Learning SDK (p. 176) to
interact with an ML Image Classiﬁcation connector.

Note

You can download the SDK from the AWS IoT Greengrass Machine Learning SDK (p. 23)
downloads page.
The example initializes an SDK client and synchronously calls the SDK's invoke_inference_service
function to invoke the local inference service. It passes in the algorithm type, service name, image
type, and image content. Then, the example parses the service response to get the probability results
(predictions).
Python 3.7
import logging
from threading import Timer
import numpy as np
import greengrass_machine_learning_sdk as ml
# We assume the inference input image is provided as a local file
# to this inference client Lambda function.
with open('/test_img/test.jpg', 'rb') as f:
content = bytearray(f.read())
client = ml.client('inference')
def infer():
logging.info('invoking Greengrass ML Inference service')
try:

resp = client.invoke_inference_service(
AlgoType='image-classification',
ServiceName='imageClassification',
ContentType='image/jpeg',
Body=content
)
except ml.GreengrassInferenceException as e:
logging.info('inference exception {}("{}")'.format(e.__class__.__name__, e))
return
except ml.GreengrassDependencyException as e:
logging.info('dependency exception {}("{}")'.format(e.__class__.__name__, e))
return
logging.info('resp: {}'.format(resp))
predictions = resp['Body'].read().decode("utf-8")
logging.info('predictions: {}'.format(predictions))
# The connector output is in the format: [0.3,0.1,0.04,...]
# Remove the '[' and ']' at the beginning and end.
predictions = predictions[1:-1]

334


count = len(predictions.split(','))
predictions_arr = np.fromstring(predictions, count=count, sep=',')

# Perform business logic that relies on the predictions_arr, which is an array
# of probabilities.

# Schedule the infer() function to run again in one second.
Timer(1, infer).start()

return

infer()

def function_handler(event, context):
    return

Python 2.7

import logging
from threading import Timer
import numpy as np
import greengrass_machine_learning_sdk as ml

# We assume the inference input image is provided as a local file
# to this inference client Lambda function.
with open('/test_img/test.jpg', 'rb') as f:
    content = f.read()

client = ml.client('inference')

def infer():
    logging.info('invoking Greengrass ML Inference service')
    try:
        resp = client.invoke_inference_service(
            AlgoType='image-classification',
            ServiceName='imageClassification',
            ContentType='image/jpeg',
            Body=content
        )
    except ml.GreengrassInferenceException as e:
        logging.info('inference exception {}:{}'.format(e.__class__.__name__, e))
        return
    except ml.GreengrassDependencyException as e:
        logging.info('dependency exception {}:{}'.format(e.__class__.__name__, e))
        return

    logging.info('resp: {}'.format(resp))
    predictions = resp['Body'].read()
    predictions = predictions[1:-1]
    count = len(predictions.split(','))
predictions_arr = np.fromstring(predictions, count=count, sep=',')

    # Perform business logic that relies on the predictions_arr, which is an array
    # of probabilities.

    # Schedule the infer() function to run again in one second.
    Timer(1, infer).start()

    return
The `infer` function

```python
def function_handler(event, context):
    return
```

The `invoke_inference_service` function in the AWS IoT Greengrass Machine Learning SDK accepts the following arguments.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlgoType</td>
<td>The name of the algorithm type to use for inference. Currently, only <code>image-classification</code> is supported.</td>
</tr>
<tr>
<td></td>
<td>Required: true</td>
</tr>
<tr>
<td></td>
<td>Type: string</td>
</tr>
<tr>
<td></td>
<td>Valid values: <code>image-classification</code></td>
</tr>
<tr>
<td>ServiceName</td>
<td>The name of the local inference service. Use the name that you specified for the <code>LocalInferenceServiceName</code> parameter when you configured the connector.</td>
</tr>
<tr>
<td></td>
<td>Required: true</td>
</tr>
<tr>
<td></td>
<td>Type: string</td>
</tr>
<tr>
<td>ContentType</td>
<td>The mime type of the input image.</td>
</tr>
<tr>
<td></td>
<td>Required: true</td>
</tr>
<tr>
<td></td>
<td>Type: string</td>
</tr>
<tr>
<td></td>
<td>Valid values: <code>image/jpeg</code>, <code>image/png</code></td>
</tr>
<tr>
<td>Body</td>
<td>The content of the input image file.</td>
</tr>
<tr>
<td></td>
<td>Required: true</td>
</tr>
<tr>
<td></td>
<td>Type: binary</td>
</tr>
</tbody>
</table>

**Installing MXNet Dependencies on the AWS IoT Greengrass Core**

To use an ML Image Classification connector, you must install the dependencies for the Apache MXNet framework on the core device. The connectors use the framework to serve the ML model.

**Note**

These connectors are bundled with a precompiled MXNet library, so you don't need to install the MXNet framework on the core device.

AWS IoT Greengrass provides scripts to install the dependencies for the following common platforms and devices (or to use as a reference for installing them). If you're using a different platform or device, see the MXNet documentation for your configuration.

Before installing the MXNet dependencies, make sure that the required system libraries (p. 340) (with the specified minimum versions) are present on the device.
NVIDIA Jetson TX2

1. Install CUDA Toolkit 9.0 and cuDNN 7.0. You can follow the instructions in the section called "Setting Up Other Devices" (p. 75) in the Getting Started tutorial.

2. Enable universe repositories so the connector can install community-maintained open software. For more information, see Repositories/Ubuntu in the Ubuntu documentation.
   a. Open the /etc/apt/sources.list file.
   b. Make sure that the following lines are uncommented.

```
deb http://ports.ubuntu.com/ubuntu-ports/ xenial universe
deb-src http://ports.ubuntu.com/ubuntu-ports/ xenial universe
deb http://ports.ubuntu.com/ubuntu-ports/ xenial-updates universe
deb-src http://ports.ubuntu.com/ubuntu-ports/ xenial-updates universe
```

3. Save a copy of the following installation script to a file named nvidiajtx2.sh on the core device.

---

**Python 3.7**

```
#!/bin/bash
set -e

echo "Installing dependencies on the system..."
echo 'Assuming that universe repos are enabled and checking dependencies...'
apt-get -y update
apt-get -y dist-upgrade
apt-get install -y liblapack3 libopenblas-dev liblapack-dev libblas-base-dev
apt-get install --python3 python3.7 python3.7-dev

python3.7 -m pip install --upgrade pip
python3.7 -m pip install numpy==1.15.0
python3.7 -m pip install opencv-python

if pip install opencv-python || echo 'Error: Unable to install OpenCV with pip on this platform. Try building the latest OpenCV from source (https://github.com/opencv/opencv).'

then
done
else
    echo 'Dependency installation/upgrade complete.'
fi
```

**Note**

If OpenCV does not install successfully using this script, you can try building from source. For more information, see Installation in Linux in the OpenCV documentation, or refer to other online resources for your platform.

---

**Python 2.7**

```
#!/bin/bash
set -e

echo "Installing dependencies on the system..."
echo 'Assuming that universe repos are enabled and checking dependencies...'
apt-get -y update
apt-get -y dist-upgrade
apt-get install -y liblapack3 libopenblas-dev liblapack-dev libblas-base-dev
python-dev
cd /usr/local/
pip install numpy==1.15.0

if pip install numpy==1.15.0
    then
        echo 'Installing latest pip...'
wget https://bootstrap.pypa.io/get-pip.py
python get-pip.py
rm get-pip.py

    then
    pip install numpy==1.15.0 scipy
```

---

337
4. From the directory where you saved the file, run the following command:

```bash
sudo nvidiajtx2.sh
```

x86_64 (Ubuntu or Amazon Linux)

1. Save a copy of the following installation script to a file named `x86_64.sh` on the core device.

Python 3.7

```bash
#!/bin/bash
set -e

echo "Installing dependencies on the system..."

release=$(awk -F= '/^NAME/{print $2}' /etc/os-release)

if [ "$release" == "Ubuntu" ]; then
    # Ubuntu. Supports EC2 and DeepLens. DeepLens has all the dependencies installed, so
    # this is mostly to prepare dependencies on Ubuntu EC2 instance.
    apt-get -y update
    apt-get -y dist-upgrade
    apt-get install -y libgfortran3 libsm6 libxext6 libxrender1
    apt-get install -y python3.7 python3.7-dev
elif [ "$release" == "Amazon Linux" ]; then
    # Amazon Linux. Expect python to be installed already
    yum -y update
    yum -y upgrade
    yum install -y compat-gcc-48-libgfortran libSM libXrender libXext
else
    echo "OS Release not supported: $release"
    exit 1
fi

python3.7 -m pip install --upgrade pip
python3.7 -m pip install numpy==1.15.0
python3.7 -m pip install opencv-python || echo 'Error: Unable to install OpenCV with pip on this platform. Try building the latest OpenCV from source (https://github.com/opencv/opencv).'

echo 'Dependency installation/upgrade complete.'
```

**Note**

If OpenCV does not install successfully using this script, you can try building from source. For more information, see Installation in Linux in the OpenCV documentation, or refer to other online resources for your platform.

Python 2.7

```bash
#!/bin/bash
set -e

echo "Installing dependencies on the system..."

release=$(awk -F= '/^NAME/{print $2}' /etc/os-release)
```
if [ "$release" == "Ubuntu" ]; then
  # Ubuntu. Supports EC2 and DeepLens. DeepLens has all the dependencies
  # installed, so
  # this is mostly to prepare dependencies on Ubuntu EC2 instance.
  apt-get -y update
  apt-get -y dist-upgrade

  apt-get install -y libgfortran3 libsm6 libxext6 libxrender1 python-dev python-pip
elif [ "$release" == "Amazon Linux" ]; then
  # Amazon Linux. Expect python to be installed already
  yum -y update
  yum -y upgrade

  yum install -y compat-gcc-48-libgfortran libSM libXrender libXext python-pip
else
  echo "OS Release not supported: $release"
  exit 1
fi

pip install numpy==1.15.0 scipy opencv-python

2. From the directory where you saved the file, run the following command:

   sudo x86_64.sh

Armv7 (Raspberry Pi)

1. Save a copy of the following installation script to a file named armv7l.sh on the core device.
   
   Python 3.7

   ```bash
   #!/bin/bash
   set -e

   echo "Installing dependencies on the system..."

   apt-get update
   apt-get -y upgrade

   apt-get install -y liblapack3 libopenblas-dev liblapack-dev
   apt-get install -y python3.7 python3.7-dev

   python3.7 -m pip install --upgrade pip
   python3.7 -m pip install numpy==1.15.0
   python3.7 -m pip install opencv-python || echo 'Error: Unable to install OpenCV
   with pip on this platform. Try building the latest OpenCV from source (https://
   github.com/opencv/opencv).'

   echo 'Dependency installation/upgrade complete.'
   ```

Note

If OpenCV does not install successfully using this script, you can try building from source. For more information, see Installation in Linux in the OpenCV documentation, or refer to other online resources for your platform.
Python 2.7

```bash
#!/bin/bash
set -e

echo "Installing dependencies on the system..."
apt-get update
apt-get -y upgrade
apt-get install -y liblapack3 libopenblas-dev liblapack-dev python-dev

# python-opencv depends on python-numpy. The latest version in the APT
# repository is python-numpy-1.8.2
# This script installs python-numpy first so that python-opencv can be
# installed, and then install the latest
# numpy-1.15.x with pip
apt-get install -y python-numpy python-opencv
dpkg --remove --force-depends python-numpy

echo 'Install latest pip...'
wget https://bootstrap.pypa.io/get-pip.py
python get-pip.py
rm get-pip.py

pip install --upgrade numpy==1.15.0 picamera scipy

echo 'Dependency installation/upgrade complete.'
```

2. From the directory where you saved the file, run the following command:

```
sudo bash armv7l.sh
```

**Note**

On a Raspberry Pi, using pip to install machine learning dependencies is a memory-intensive operation that can cause the device to run out of memory and become unresponsive. As a workaround, you can temporarily increase the swap size:

In `/etc/dphys-swapfile`, increase the value of the `CONF_SWAPSIZE` variable and then run the following command to restart `dphys-swapfile`.

```
/etc/init.d/dphys-swapfile restart
```

Logging and Troubleshooting

Depending on your group settings, event and error logs are written to CloudWatch Logs, the local file system, or both. Logs from this connector use the prefix `LocalInferenceServiceName`. If the connector behaves unexpectedly, check the connector's logs. These usually contain useful debugging information, such as a missing ML library dependency or the cause of a connector startup failure.

If the AWS IoT Greengrass group is configured to write local logs, the connector writes log files to `greengrass-root/ggc/var/log/user/region/aws/`. For more information about Greengrass logging, see Monitoring (p. 457).

Use the following information to help troubleshoot issues with the ML Image Classification connectors.

**Required system libraries**

The following tabs list the system libraries required for each ML Image Classification connector.
### ML Image Classification Aarch64 JTX2

<table>
<thead>
<tr>
<th>Library</th>
<th>Minimum version</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld-linux-aarch64.so.1</td>
<td>GLIBC_2.17</td>
</tr>
<tr>
<td>libc.so.6</td>
<td>GLIBC_2.17</td>
</tr>
<tr>
<td>libcublas.so.9.0</td>
<td>not applicable</td>
</tr>
<tr>
<td>libcudart.so.9.0</td>
<td>not applicable</td>
</tr>
<tr>
<td>libcudnn.so.7</td>
<td>not applicable</td>
</tr>
<tr>
<td>libcufft.so.9.0</td>
<td>not applicable</td>
</tr>
<tr>
<td>libcurand.so.9.0</td>
<td>not applicable</td>
</tr>
<tr>
<td>libcusolver.so.9.0</td>
<td>not applicable</td>
</tr>
<tr>
<td>libgcc_s.so.1</td>
<td>GCC_4.2.0</td>
</tr>
<tr>
<td>libgomp.so.1</td>
<td>GOMP_4.0, OMP_1.0</td>
</tr>
<tr>
<td>libm.so.6</td>
<td>GLIBC_2.23</td>
</tr>
<tr>
<td>libpthread.so.0</td>
<td>GLIBC_2.17</td>
</tr>
<tr>
<td>librt.so.1</td>
<td>GLIBC_2.17</td>
</tr>
<tr>
<td>libstdc++.so.6</td>
<td>GLIBCXX_3.4.21, CXXABI_1.3.8</td>
</tr>
</tbody>
</table>

### ML Image Classification x86_64

<table>
<thead>
<tr>
<th>Library</th>
<th>Minimum version</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld-linux-x86-64.so.2</td>
<td>GCC_4.0.0</td>
</tr>
<tr>
<td>libc.so.6</td>
<td>GLIBC_2.4</td>
</tr>
<tr>
<td>libgfortran.so.3</td>
<td>GFORTRAN_1.0</td>
</tr>
<tr>
<td>libm.so.6</td>
<td>GLIBC_2.23</td>
</tr>
<tr>
<td>libpthread.so.0</td>
<td>GLIBC_2.2.5</td>
</tr>
<tr>
<td>librt.so.1</td>
<td>GLIBC_2.2.5</td>
</tr>
<tr>
<td>libstdc++.so.6</td>
<td>CXXABI_1.3.8, GLIBCXX_3.4.21</td>
</tr>
</tbody>
</table>

### ML Image Classification Armv7

<table>
<thead>
<tr>
<th>Library</th>
<th>Minimum version</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld-linux-armhf.so.3</td>
<td>GLIBC_2.4</td>
</tr>
<tr>
<td>libc.so.6</td>
<td>GLIBC_2.7</td>
</tr>
<tr>
<td>libgcc_s.so.1</td>
<td>GCC_4.0.0</td>
</tr>
</tbody>
</table>
### Issues

<table>
<thead>
<tr>
<th>Symptom</th>
<th>Solution</th>
</tr>
</thead>
</table>
| On a Raspberry Pi, the following error message is logged and you are not using the camera: Failed to initialize libdc1394 | Run the following command to disable the driver:  
  `sudo ln /dev/null /dev/raw1394`  
  This operation is ephemeral and the symbolic link will disappear after rebooting. Consult the manual of your OS distribution to learn how to automatically create the link up on reboot. |

### Licenses

The ML Image Classification connectors includes the following third-party software/licensing:

- AWS SDK for Python (Boto 3)/Apache 2.0
- Deep Neural Network Library (DNNL)/Apache 2.0
- OpenMP® Runtime Library/See Intel OpenMP Runtime Library licensing (p. 342).
- mxnet/Apache 2.0
- six/MIT

**Intel OpenMP Runtime Library licensing.** The Intel® OpenMP® runtime is dual-licensed, with a commercial (COM) license as part of the Intel® Parallel Studio XE Suite products, and a BSD open source (OSS) license. For more information, see Licensing in the Intel® OpenMP® Runtime Library documentation.

This connector is released under the Greengrass Core Software License Agreement.

### Changelog

The following table describes the changes in each version of the connector.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Added the MLFeedbackConnectorConfigId parameter to support the use of the ML Feedback connector (p. 315) to upload model input data,</td>
</tr>
</tbody>
</table>
**ML Object Detection Connector**

The ML Object Detection connectors (p. 283) provide a machine learning (ML) inference service that runs on the AWS IoT Greengrass core. This local inference service performs object detection using an object detection model compiled by the Amazon SageMaker Neo deep learning compiler. Two types of object detection models are supported: Single Shot Multibox Detector (SSD) and You Only Look Once (YOLO) v3. For more information, see Object Detection Model Requirements (p. 344).

User-defined Lambda functions use the AWS IoT Greengrass Machine Learning SDK to submit inference requests to the local inference service. The service performs local inference on an input image and returns a list of predictions for each object detected in the image. Each prediction contains an object category, a prediction confidence score, and pixel coordinates that specify a bounding box around the predicted object.

AWS IoT Greengrass provides ML Object Detection connectors for multiple platforms:

<table>
<thead>
<tr>
<th>Connector</th>
<th>Description and ARN</th>
</tr>
</thead>
<tbody>
<tr>
<td>ML Object Detection Aarch64 JTX2</td>
<td>Object detection inference service for NVIDIA Jetson TX2. Supports GPU acceleration.</td>
</tr>
<tr>
<td></td>
<td><strong>ARN:</strong> arn:aws:greengrass:region::connectors/ObjectDetectionAarch64JTX2/versions/1</td>
</tr>
<tr>
<td>ML Object Detection x86_64</td>
<td>Object detection inference service for x86_64 platforms.</td>
</tr>
<tr>
<td></td>
<td><strong>ARN:</strong> arn:aws:greengrass:region::connectors/ObjectDetectionx86-64/versions/1</td>
</tr>
<tr>
<td>ML Object Detection ARMv7</td>
<td>Object detection inference service for ARMv7 platforms.</td>
</tr>
<tr>
<td></td>
<td><strong>ARN:</strong> arn:aws:greengrass:region::connectors/ObjectDetectionARMv7/versions/1</td>
</tr>
</tbody>
</table>
Requirements

These connectors have the following requirements:

- AWS IoT Greengrass Core Software v1.9.3 or later.
- Python version 3.7 installed on the core device and added to the PATH environment variable.
- Dependencies for the Amazon SageMaker Neo deep learning runtime installed on the core device. For more information, see the section called “Installing Neo Deep Learning Runtime Dependencies” (p. 349).
- An ML resource (p. 221) in the Greengrass group. The ML resource must reference an Amazon S3 bucket that contains an object detection model. For more information, see Amazon S3 model sources (p. 223).
  
  **Note**
  
  The model must be a Single Shot Multibox Detector or You Only Look Once v3 object detection model type. It must be compiled using the Amazon SageMaker Neo deep learning compiler. For more information, see Object Detection Model Requirements (p. 344).
- The ML Feedback connector (p. 315) added to the Greengrass group and configured. This is required only if you want to use the connector to upload model input data and publish predictions to an MQTT topic.
- AWS IoT Greengrass Machine Learning SDK (p. 176) v1.1.0 is required to interact with this connector.

Object Detection Model Requirements

The ML Object Detection connectors support Single Shot multibox Detector (SSD) and You Only Look Once (YOLO) v3 object detection model types. You can use the object detection components provided by GluonCV to train the model with your own dataset. Or, you can use pre-trained models from the GluonCV Model Zoo:

- Pre-trained SSD model
- Pre-trained YOLO v3 model

Your object detection model must be trained with 512 x 512 input images. The pre-trained models from the GluonCV Model Zoo already meet this requirement.

Trained object detection models must be compiled with the Amazon SageMaker Neo deep learning compiler. When compiling, make sure the target hardware matches the hardware of your Greengrass core device. For more information, see Amazon SageMaker Neo in the Amazon SageMaker Developer Guide.

The compiled model must be added as an ML resource (Amazon S3 model source (p. 223)) to the same Greengrass group as the connector.

Connector Parameters

These connectors provide the following parameters.

**MLModelDestinationPath**

The absolute path to the the Amazon S3 bucket that contains the Neo-compatible ML model. This is the destination path that's specified for the ML model resource.

Display name in console: **Model destination path**

Required: true

Type: string
Valid pattern: .+
MLModelResourceId

The ID of the ML resource that references the source model.
Display name in console: Greengrass group ML resource
Required: true
Type: S3MachineLearningModelResource
Valid pattern: ^[a-zA-Z0-9:_-]+$ LocalInferenceServiceName

The name for the local inference service. User-defined Lambda functions invoke the service by passing the name to the invoke_inference_service function of the AWS IoT Greengrass Machine Learning SDK. For an example, see the section called “Usage Example” (p. 347).
Display name in console: Local inference service name
Required: true
Type: string
Valid pattern: ^[a-zA-Z0-9][a-zA-Z0-9-]{1,62}$ LocalInferenceServiceTimeoutSeconds

The time (in seconds) before the inference request is terminated. The minimum value is 1. The default value is 10.
Display name in console: Timeout (second)
Required: true
Type: string
Valid pattern:^[1-9][0-9]*$ LocalInferenceServiceMemoryLimitKB

The amount of memory (in KB) that the service has access to. The minimum value is 1.
Display name in console: Memory limit
Required: true
Type: string
Valid pattern:^[1-9][0-9]*$ GPUAcceleration

The CPU or GPU (accelerated) computing context. This property applies to the ML Image Classification Aarch64 JTX2 connector only.
Display name in console: GPU acceleration
Required: true
Type: string
Valid values: CPU or GPU MLPFeedbackConnectorConfigId

The ID of the feedback configuration to use to upload model input data. This must match the ID of a feedback configuration defined for the ML Feedback connector (p. 315).
This parameter is required only if you want to use the ML Feedback connector to upload model input data and publish predictions to an MQTT topic.

Display name in console: **ML Feedback connector configuration ID**

Required: false

Type: string

Valid pattern: ^\$|^[a-zA-Z0-9][a-zA-Z0-9-]{1,62}$

**Create Connector Example (CLI)**

The following CLI command creates a **ConnectorDefinition** with an initial version that contains an ML Object Detection connector. This example creates an instance of the ML Object Detection ARMv7l connector.

```
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-version ' {
  "Connectors": [
    {
      "Id": "MyObjectDetectionConnector",
      "ConnectorArn": "arn:aws:greengrass:region::/connectors/ObjectDetectionARMv7/versions/1",
      "Parameters": {
        "MLModelDestinationPath": "/path-to-model",
        "MLModelResourceId": "my-ml-resource",
        "LocalInferenceServiceName": "objectDetection",
        "LocalInferenceServiceTimeoutSeconds": "10",
        "LocalInferenceServiceMemoryLimitKB": "500000",
        "MLFeedbackConnectorConfigId": "object-detector-random-sampling"
      }
    }
  ]
}
```

**Note**

The Lambda function in these connectors have a long-lived (p. 187) lifecycle.

In the AWS IoT Greengrass console, you can add a connector from the group's **Connectors** page. For more information, see the section called "Get Started with Connectors (Console)" (p. 403).

**Input Data**

These connectors accept an image file as input. Input image files must be in jpeg or png format. For more information, see the section called "Usage Example" (p. 347).

These connectors don't accept MQTT messages as input data.

**Output Data**

These connectors return a formatted list of prediction results for the identified objects in the input image:

```
{
  "prediction": [
    [14,
     0.9384938478469849,
     0.37763649225234985],
```

346
Each prediction in the list is contained in square brackets and contains six values:

- The first value represents the predicted object category for the identified object. Object categories and their corresponding values are determined when training your object detection machine learning model in the Neo deep learning compiler.
- The second value is the confidence score for the object category prediction. This represents the probability that the prediction was correct.
- The last four values correspond to pixel dimensions that represent a bounding box around the predicted object in the image.

These connectors don’t publish MQTT messages as output data.

**Usage Example**

The following example Lambda function uses the [AWS IoT Greengrass Machine Learning SDK](p. 176) to interact with an ML Object Detection connector.

**Note**

You can download the SDK from the [AWS IoT Greengrass Machine Learning SDK](p. 23) downloads page.

The example initializes an SDK client and synchronously calls the SDK’s `invoke_inference_service` function to invoke the local inference service. It passes in the algorithm type, service name, image type, and image content. Then, the example parses the service response to get the probability results (predictions).

```python
import logging
from threading import Timer
import numpy as np
import greengrass_machine_learning_sdk as ml

# We assume the inference input image is provided as a local file
# to this inference client Lambda function.
with open('/test_img/test.jpg', 'rb') as f:
```
content = bytearray(f.read())

client = ml.client('inference')

def infer():
    logging.info('invoking Greengrass ML Inference service')
    try:
        resp = client.invoke_inference_service(
            AlgoType='object-detection',
            ServiceName='objectDetection',
            ContentType='image/jpeg',
            Body=content
        )
    except ml.GreengrassInferenceException as e:
        logging.info('inference exception {}'.format(e.__class__.__name__, e))
        return
    except ml.GreengrassDependencyException as e:
        logging.info('dependency exception {}'.format(e.__class__.__name__, e))
        return

    logging.info('resp: {}'.format(resp))
    predictions = resp['Body'].read().decode("utf-8")
    logging.info('predictions: {}'.format(predictions_str))
    predictions = eval(predictions_str)
    # Perform business logic that relies on the predictions.
    # Schedule the infer() function to run again in ten second.
    Timer(10, infer).start()
    return

infer()

def function_handler(event, context):
    return

The `invoke_inference_service` function in the AWS IoT Greengrass Machine Learning SDK accepts the following arguments.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlgoType</td>
<td>The name of the algorithm type to use for inference. Currently, only object-detection is supported.</td>
</tr>
<tr>
<td></td>
<td>Required: true</td>
</tr>
<tr>
<td></td>
<td>Type: string</td>
</tr>
<tr>
<td></td>
<td>Valid values: object-detection</td>
</tr>
<tr>
<td>ServiceName</td>
<td>The name of the local inference service. Use the name that you specified for the LocalInferenceServiceName parameter when you configured the connector.</td>
</tr>
<tr>
<td></td>
<td>Required: true</td>
</tr>
<tr>
<td></td>
<td>Type: string</td>
</tr>
<tr>
<td>ContentType</td>
<td>The mime type of the input image.</td>
</tr>
</tbody>
</table>
### Argument Table

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Required: true</td>
</tr>
<tr>
<td></td>
<td>Type: string</td>
</tr>
<tr>
<td></td>
<td>Valid values: image/jpeg, image/png</td>
</tr>
<tr>
<td>Body</td>
<td>The content of the input image file.</td>
</tr>
<tr>
<td></td>
<td>Required: true</td>
</tr>
<tr>
<td></td>
<td>Type: binary</td>
</tr>
</tbody>
</table>

## Installing Neo Deep Learning Runtime Dependencies on the AWS IoT Greengrass Core

The ML Object Detection connectors are bundled with the Amazon SageMaker Neo deep learning runtime (DLR). The connectors use the runtime to serve the ML model. To use these connectors, you must install the dependencies for the DLR on your core device.

Before you install the DLR dependencies, make sure that the required system libraries (p. 351) (with the specified minimum versions) are present on the device.

### NVIDIA Jetson TX2

1. Install CUDA Toolkit 9.0 and cuDNN 7.0. You can follow the instructions in the section called "Setting Up Other Devices" (p. 75) in the Getting Started tutorial.

2. Enable universe repositories so the connector can install community-maintained open software. For more information, see Repositories/Ubuntu in the Ubuntu documentation.
   
   a. Open the `/etc/apt/sources.list` file.
   b. Make sure that the following lines are uncommented.

   ```
   deb http://ports.ubuntu.com/ubuntu-ports/ xenial universe
   deb-src http://ports.ubuntu.com/ubuntu-ports/ xenial universe
   deb http://ports.ubuntu.com/ubuntu-ports/ xenial-updates universe
   deb-src http://ports.ubuntu.com/ubuntu-ports/ xenial-updates universe
   ```

3. Save a copy of the following installation script to a file named `nvidiajtx2.sh` on the core device.

```bash
#!/bin/bash
set -e

echo "Installing dependencies on the system..."
echo 'Assuming that universe repos are enabled and checking dependencies...'
apt-get -y update
apt-get -y dist-upgrade
apt-get install -y liblapack3 libopenblas-dev liblapack-dev libatlas-base-dev
apt-get install -y python3.7 python3.7-dev

python3.7 -m pip install --upgrade pip
python3.7 -m pip install numpy==1.15.0
python3.7 -m pip install opencv-python || echo 'Error: Unable to install OpenCV with pip on this platform. Try building the latest OpenCV from source (https://github.com/opencv/opencv).'
```
echo 'Dependency installation/upgrade complete.'

**Note**
If OpenCV does not install successfully using this script, you can try building from source. For more information, see Installation in Linux in the OpenCV documentation, or refer to other online resources for your platform.

4. From the directory where you saved the file, run the following command:

    sudo nvidiajtx2.sh

x86_64 (Ubuntu or Amazon Linux)

1. Save a copy of the following installation script to a file named `x86_64.sh` on the core device.

    ```bash
    #!/bin/bash
    set -e

    echo "Installing dependencies on the system..."

    release=$(awk -F= '/^NAME/{print $2}' /etc/os-release)
    if [ "$release" == "Ubuntu" ]; then
        # Ubuntu. Supports EC2 and DeepLens. DeepLens has all the dependencies installed, so
        # this is mostly to prepare dependencies on Ubuntu EC2 instance.
        apt-get -y update
        apt-get -y dist-upgrade

        apt-get install -y libgfortran3 libsm6 libxext6 libxrender1
        apt-get install -y python3.7 python3.7-dev
    elif [ "$release" == "Amazon Linux" ]; then
        # Amazon Linux. Expect python to be installed already
        yum -y update
        yum -y upgrade

        yum install -y compat-gcc-48-libgfortran libSM libXrender libXext
    else
        echo "OS Release not supported: $release"
        exit 1
    fi

    python3.7 -m pip install --upgrade pip
    python3.7 -m pip install numpy==1.15.0
    python3.7 -m pip install opencv-python || echo 'Error: Unable to install OpenCV with pip on this platform. Try building the latest OpenCV from source (https://github.com/opencv/opencv).'

    echo 'Dependency installation/upgrade complete.'

**Note**
If OpenCV does not install successfully using this script, you can try building from source. For more information, see Installation in Linux in the OpenCV documentation, or refer to other online resources for your platform.

2. From the directory where you saved the file, run the following command:

    sudo x86_64.sh
ARMv7 (Raspberry Pi)

1. Save a copy of the following installation script to a file named `armv7l.sh` on the core device.

```
#!/bin/bash
set -e

echo "Installing dependencies on the system..."

apt-get update
apt-get -y upgrade
apt-get install -y liblapack3 libopenblas-dev liblapack-dev
apt-get install -y python3.7 python3.7-dev

python3.7 -m pip install --upgrade pip
python3.7 -m pip install numpy==1.15.0
python3.7 -m pip install opencv-python || echo 'Error: Unable to install OpenCV
with pip on this platform. Try building the latest OpenCV from source (https://
github.com/opencv/opencv).'

echo 'Dependency installation/upgrade complete.'
```

**Note**

If OpenCV does not install successfully using this script, you can try building from
source. For more information, see Installation in Linux in the OpenCV documentation,
or refer to other online resources for your platform.

2. From the directory where you saved the file, run the following command:

```
sudo bash armv7l.sh
```

**Note**

On a Raspberry Pi, using `pip` to install machine learning dependencies is a memory-intensive operation that can cause the device to run out of memory and become unresponsive. As a workaround, you can temporarily increase the swap size. In `/etc/dphys-swapfile`, increase the value of the `CONF_SWAPSIZE` variable and then run the following command to restart `dphys-swapfile`.

```
/etc/init.d/dphys-swapfile restart
```

### Logging and Troubleshooting

Depending on your group settings, event and error logs are written to CloudWatch Logs, the local file system, or both. Logs from this connector use the prefix `LocalInferenceServiceName`. If the connector behaves unexpectedly, check the connector's logs. These usually contain useful debugging information, such as a missing ML library dependency or the cause of a connector startup failure.

If the AWS IoT Greengrass group is configured to write local logs, the connector writes log files to `greengrass-root/ggc/var/log/user/region/aws/`. For more information about Greengrass logging, see Monitoring (p. 457).

Use the following information to help troubleshoot issues with the ML Object Detection connectors.

#### Required system libraries

The following tabs list the system libraries required for each ML Object Detection connector.
ML Object Detection Aarch64 JTX2

<table>
<thead>
<tr>
<th>Library</th>
<th>Minimum version</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld-linux-aarch64.so.1</td>
<td>GLIBC_2.17</td>
</tr>
<tr>
<td>libc.so.6</td>
<td>GLIBC_2.17</td>
</tr>
<tr>
<td>libcublas.so.9.0</td>
<td>not applicable</td>
</tr>
<tr>
<td>libcudart.so.9.0</td>
<td>not applicable</td>
</tr>
<tr>
<td>libcudnn.so.7</td>
<td>not applicable</td>
</tr>
<tr>
<td>libcufft.so.9.0</td>
<td>not applicable</td>
</tr>
<tr>
<td>libcudnn.so.9.0</td>
<td>not applicable</td>
</tr>
<tr>
<td>libnvvi.so.1</td>
<td>GCC_4.2.0</td>
</tr>
<tr>
<td>libgomp.so.1</td>
<td>GOMP_4.0, OMP_1.0</td>
</tr>
<tr>
<td>libm.so.6</td>
<td>GLIBC_2.23</td>
</tr>
<tr>
<td>libnvinfer.so.4</td>
<td>not applicable</td>
</tr>
<tr>
<td>libnvrm_gpu.so</td>
<td>not applicable</td>
</tr>
<tr>
<td>libnvrm.so</td>
<td>not applicable</td>
</tr>
<tr>
<td>libnvdiav-fatbinaryloader.so.28.2.1</td>
<td>not applicable</td>
</tr>
<tr>
<td>libnvos.so</td>
<td>not applicable</td>
</tr>
<tr>
<td>libpthread.so.0</td>
<td>GLIBC_2.17</td>
</tr>
<tr>
<td>librt.so.1</td>
<td>GLIBC_2.17</td>
</tr>
<tr>
<td>libstdc++.so.6</td>
<td>GLIBCXX_3.4.21, CXXABI_1.3.8</td>
</tr>
</tbody>
</table>

ML Object Detection x86_64

<table>
<thead>
<tr>
<th>Library</th>
<th>Minimum version</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld-linux-x86-64.so.2</td>
<td>GCC_4.0.0</td>
</tr>
<tr>
<td>libc.so.6</td>
<td>GLIBC_2.4</td>
</tr>
<tr>
<td>libgfortran.so.3</td>
<td>GFORTRAN_1.0</td>
</tr>
<tr>
<td>libm.so.6</td>
<td>GLIBC_2.23</td>
</tr>
<tr>
<td>libpthread.so.0</td>
<td>GLIBC_2.2.5</td>
</tr>
<tr>
<td>librt.so.1</td>
<td>GLIBC_2.2.5</td>
</tr>
<tr>
<td>libstdc++.so.6</td>
<td>CXXABI_1.3.8, GLIBCXX_3.4.21</td>
</tr>
</tbody>
</table>
ML Object Detection ARMv7

<table>
<thead>
<tr>
<th>Library</th>
<th>Minimum version</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld-linux-armhf.so.3</td>
<td>GLIBC_2.4</td>
</tr>
<tr>
<td>libc.so.6</td>
<td>GLIBC_2.7</td>
</tr>
<tr>
<td>libgcc_s.so.1</td>
<td>GCC_4.0.0</td>
</tr>
<tr>
<td>libgfortran.so.3</td>
<td>GFORTRAN_1.0</td>
</tr>
<tr>
<td>libm.so.6</td>
<td>GLIBC_2.4</td>
</tr>
<tr>
<td>libpthread.so.0</td>
<td>GLIBC_2.4</td>
</tr>
<tr>
<td>librt.so.1</td>
<td>GLIBC_2.4</td>
</tr>
<tr>
<td>libstdc++.so.6</td>
<td>CXXABI_1.3.8, CXXABI_ARM_1.3.3,</td>
</tr>
<tr>
<td></td>
<td>GLIBCXX_3.4.20</td>
</tr>
</tbody>
</table>

**Issues**

<table>
<thead>
<tr>
<th>Symptom</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>On a Raspberry Pi, the following error message is logged and you are not using the camera: Failed to initialize libdc1394</td>
<td>Run the following command to disable the driver:</td>
</tr>
<tr>
<td></td>
<td>sudo ln /dev/null /dev/raw1394</td>
</tr>
<tr>
<td></td>
<td>This operation is ephemeral. The symbolic link disappears after you reboot. Consult the manual of your OS distribution to learn how to create the link automatically upon reboot.</td>
</tr>
</tbody>
</table>

**Licenses**

The ML Object Detection connectors include the following third-party software/licensing:

- AWS SDK for Python (Boto 3)/Apache 2.0
- Deep Learning Runtime/Apache 2.0
- six/MIT

This connector is released under the Greengrass Core Software License Agreement.

**See Also**

- Integrate with Services and Protocols Using Connectors (p. 283)
- the section called “Get Started with Connectors (Console)” (p. 403)
- the section called “Get Started with Connectors (CLI)” (p. 413)
- Perform Machine Learning Inference (p. 221)
- Object Detection Algorithm in the Amazon SageMaker Developer Guide
Modbus-RTU Protocol Adapter Connector

The Modbus-RTU Protocol Adapter connector (p. 283) polls information from Modbus RTU devices that are in the AWS IoT Greengrass group.

This connector receives parameters for a Modbus RTU request from a user-defined Lambda function. It sends the corresponding request, and then publishes the response from the target device as an MQTT message.

This connector has the following versions.

<table>
<thead>
<tr>
<th>Version</th>
<th>ARN</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>arn:aws:greengrass:region::/connectors/ModbusRTUProtocolAdapter/versions/2</td>
</tr>
<tr>
<td>1</td>
<td>arn:aws:greengrass:region::/connectors/ModbusRTUProtocolAdapter/versions/1</td>
</tr>
</tbody>
</table>

For information about version changes, see the Changelog (p. 364).

Requirements

This connector has the following requirements:

- AWS IoT Greengrass Core Software v1.7 or later.
- Python version 2.7 installed on the core device and added to the PATH environment variable.
- A physical connection between the AWS IoT Greengrass core and the Modbus devices. The core must be physically connected to the Modbus RTU network through a serial port (for example, a USB port).
- A local device resource (p. 200) in the Greengrass group that points to the physical Modbus serial port.
- A user-defined Lambda function that sends Modbus RTU request parameters to this connector. The request parameters must conform to expected patterns and include the IDs and addresses of the target devices on the Modbus RTU network. For more information, see the section called “Input Data” (p. 355).

Connector Parameters

This connector supports the following parameters:

ModbusSerialPort-ResourceId

The ID of the local device resource that represents the physical Modbus serial port.

**Note**

This connector is granted read-write access to the resource.

Display name in console: **Modbus serial port resource**

Required: true

Type: string

Valid pattern: .+
ModbusSerialPort

The absolute path to the physical Modbus serial port on the device. This is the source path that’s specified for the Modbus local device resource.

Display name in console: **Source path of Modbus serial port resource**

Required: true

Type: string

Valid pattern: .+

Create Connector Example (CLI)

The following CLI command creates a **ConnectorDefinition** with an initial version that contains the Modbus-RTU Protocol Adapter connector.

```
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-version '
  
  "Connectors": [
    
    
    "Id": "MyModbusRTUProtocolAdapterConnector",
    "ConnectorArn": "arn:aws:greengrass:region::/connectors/ModbusRTUProtocolAdapter/versions/2",
    "Parameters": {
      "ModbusSerialDevice-ResourceId": "MyLocalModbusSerialPort",
      "ModbusSerialDevice": "/path-to-port"
    }
  ]
',
```

**Note**
The Lambda function in this connector has a long-lived (p. 187) lifecycle.

In the AWS IoT Greengrass console, you can add a connector from the group’s **Connectors** page. For more information, see the section called “Get Started with Connectors (Console)” (p. 403).

**Note**
After you deploy the Modbus-RTU Protocol Adapter connector, you can use AWS IoT Things Graph to orchestrate interactions between devices in your group. For more information, see Modbus in the **AWS IoT Things Graph User Guide**.

Input Data

This connector accepts Modbus RTU request parameters from a user-defined Lambda function on an MQTT topic. Input messages must be in JSON format.

**Topic filter**

modbus/adapter/request

**Message properties**

The request message varies based on the type of Modbus RTU request that it represents. The following properties are required for all requests:

- In the request object:
  - **operation**. The operation to execute, specified by name or function code. For example, to read coils, you can specify `ReadCoilsRequest` or `0x01`. This value must be a Unicode string.
• device. The target device of the request. This value must be between 0 – 247.

• The id property. An ID for the request. This value is used for data deduplication and is returned as is in the id property of all responses, including error responses. This value must be a Unicode string.

The other parameters to include in the request depend on the operation. All request parameters are required except the CRC, which is handled separately. For examples, see the section called “Example Requests and Responses” (p. 358).

Example input: Using operation name

```json
{
    "request": {
        "operation": "ReadCoilsRequest",
        "device": 1,
        "address": 0x01,
        "count": 1
    },
    "id": "TestRequest"
}
```

Example input: Using function code

```json
{
    "request": {
        "operation": 0x01,
        "device": 1,
        "address": 0x01,
        "count": 1
    },
    "id": "TestRequest"
}
```

For more examples, see the section called “Example Requests and Responses” (p. 358).

Output Data

This connector publishes responses to incoming Modbus RTU requests.

Topic filter

modbus/adapter/response

Message properties

The format of the response message varies based on the corresponding request and the response status. For examples, see the section called “Example Requests and Responses” (p. 358).

Note

A response for a write operation is simply an echo of the request. Although no meaningful information is returned for write responses, it's a good practice to check the status of the response.

Every response includes the following properties:

• In the response object:
  • status. The status of the request. The status can be one of the following values:
    • Success. The request was valid, sent to the Modbus RTU network, and a response was returned.
• Exception. The request was valid, sent to the Modbus RTU network, and an exception response was returned. For more information, see the section called “Response Status: Exception” (p. 362).
• No Response. The request was invalid, and the connector caught the error before the request was sent over the Modbus RTU network. For more information, see the section called “Response Status: No Response” (p. 363).
• device. The device that the request was sent to.
• operation. The request type that was sent.
• payload. The response content that was returned. If the status is No Response, this object contains only an error property with the error description (for example, “error”: “[Input/Output] No Response received from the remote unit”).
• The id property. The ID of the request, used for data deduplication.

Example output: Success

```json
{
  "response" : {
    "status" : "success",
    "device": 1,
    "operation": "ReadCoilsRequest",
    "payload": {
      "function_code": 1,
      "bits": [1]
    },
    "id" : "TestRequest"
  }
}
```

Example output: Failure

```json
{
  "response" : {
    "status" : "fail",
    "error_message": "Internal Error",
    "error": "Exception",
    "device": 1,
    "operation": "ReadCoilsRequest",
    "payload": {
      "function_code": 129,
      "exception_code": 2
    },
    "id" : "TestRequest"
  }
}
```

For more examples, see the section called “Example Requests and Responses” (p. 358).

Modbus RTU Requests and Responses

This connector accepts Modbus RTU request parameters as input data (p. 355) and publishes responses as output data (p. 356).

The following common operations are supported.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Function Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReadCoilsRequest</td>
<td>01</td>
</tr>
<tr>
<td>Operation</td>
<td>Function Code</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>ReadDiscreteInputsRequest</td>
<td>02</td>
</tr>
<tr>
<td>ReadHoldingRegistersRequest</td>
<td>03</td>
</tr>
<tr>
<td>ReadInputRegistersRequest</td>
<td>04</td>
</tr>
<tr>
<td>WriteSingleCoilRequest</td>
<td>05</td>
</tr>
<tr>
<td>WriteSingleRegisterRequest</td>
<td>06</td>
</tr>
<tr>
<td>WriteMultipleCoilsRequest</td>
<td>15</td>
</tr>
<tr>
<td>WriteMultipleRegistersRequest</td>
<td>16</td>
</tr>
<tr>
<td>MaskWriteRegisterRequest</td>
<td>22</td>
</tr>
<tr>
<td>ReadWriteMultipleRegistersRequest</td>
<td>23</td>
</tr>
</tbody>
</table>

Example Requests and Responses

The following are example requests and responses for supported operations.

Read Coils

**Request example:**

```json
{
    "request": {
        "operation": "ReadCoilsRequest",
        "device": 1,
        "address": 0x01,
        "count": 1
    },
    "id": "TestRequest"
}
```

**Response example:**

```json
{
    "response": {
        "status": "success",
        "device": 1,
        "operation": "ReadCoilsRequest",
        "payload": {
            "function_code": 1,
            "bits": [1]
        }
    },
    "id": "TestRequest"
}
```

Read Discrete Inputs

**Request example:**

```json
{
    "request": {
        "operation": "ReadDiscreteInputsRequest",
        "device": 1,
        "address": 0x01,
        "count": 1
    },
    "id": "TestRequest"
}
```
"device": 1,
"address": 0x01,
"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": 0x01,
    "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": 0x01,
    "value": 1
  },
  "id": "TestRequest"
}
Write Single Coil

**Request example:**

```json
{
    "request": {
        "operation": "WriteSingleCoilRequest",
        "device": 1,
        "address": 0x01,
        "value": 1
    },
    "id": "TestRequest"
}
```

**Response example:**

```json
{
    "response": {
        "status": "success",
        "device": 1,
        "operation": "WriteSingleCoilRequest",
        "payload": {
            "function_code": 5,
            "address": 1,
            "value": true
        }
    },
    "id": "TestRequest"
}
```

Write Single Register

**Request example:**

```json
{
    "request": {
        "operation": "WriteSingleRegisterRequest",
        "device": 1,
        "address": 0x01,
        "value": 1
    },
    "id": "TestRequest"
}
```

Write Multiple Coils

**Request example:**

```json
{
    "request": {
        "operation": "WriteMultipleCoilsRequest",
        "device": 1,
        "address": 0x01,
        "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": 0x01,
        "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": 0x01,
        "and_mask": 0xaf,
        "or_mask": 0x01
    },
    "id": "TestRequest"
}
```
Response example:

```json
{
   "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:**

```json
{
   "request": {
      "operation": "ReadWriteMultipleRegistersRequest",
      "device": 1,
      "read_address": 0x01,
      "read_count": 2,
      "write_address": 0x03,
      "write_registers": [20, 30, 40]
   },
   "id": "TestRequest"
}
```

Response example:

```json
{
   "response": {
      "status": "success",
      "device": 1,
      "operation": "ReadWriteMultipleRegistersRequest",
      "payload": {
         "function_code": 23,
         "registers": [10, 20, 10, 20]
      }
   },
   "id": "TestRequest"
}
```

**Note**

The registers returned in this response are the registers that are read from.

**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:

```json
{
   "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:**

```json
{
   "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 <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.

```json
{
   "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"
}
```
Usage Example

The following example Lambda function sends an input message to the connector.

**Note**
This Python function uses the AWS IoT Greengrass Core SDK (p. 175) to publish an MQTT message. You can use the following pip command to install the Python version of the SDK on your core device:

```bash
pip install greengrasssdk
```

```
import greengrasssdk
import json

TOPIC_REQUEST = 'modbus/adapter/request'

# Creating a greengrass core sdk client
iot_client = greengrasssdk.client('iot-data')

def create_read_coils_request():
    request = {
        "request": {
            "operation": "ReadCoilsRequest",
            "device": 1,
            "address": 0x01,
            "count": 1
        },
        "id": "TestRequest"
    }
    return request

def publish_basic_request():
    iot_client.publish(payload=json.dumps(create_read_coils_request()), topic=TOPIC_REQUEST)

publish_basic_request()

def function_handler(event, context):
    return
```

Licenses

The Modbus-RTU Protocol Adapter connector includes the following third-party software/licensing:

- **pymodbus/BSD**
- **pyserial/BSD**

This connector is released under the Greengrass Core Software License Agreement.

Changelog

The following table describes the changes in each version of the connector.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Updated connector ARN for AWS Region support.</td>
</tr>
<tr>
<td></td>
<td>Improved error logging.</td>
</tr>
</tbody>
</table>
A Greengrass group can contain only one version of the connector at a time.

See Also
- Integrate with Services and Protocols Using Connectors (p. 283)
- the section called “Get Started with Connectors (Console)” (p. 403)
- the section called “Get Started with Connectors (CLI)” (p. 413)

Raspberry Pi GPIO Connector

The Raspberry Pi GPIO connector (p. 283) controls general-purpose input/output (GPIO) pins on a Raspberry Pi core device.

This connector polls input pins at a specified interval and publishes state changes to MQTT topics. It also accepts read and write requests as MQTT messages from user-defined Lambda functions. Write requests are used to set the pin to high or low voltage.

The connector provides parameters that you use to designate input and output pins. This behavior is configured before group deployment. It can't be changed at runtime.

- Input pins can be used to receive data from peripheral devices.
- Output pins can be used to control peripherals or send data to peripherals.

You can use this connector for many scenarios, such as:
- Controlling green, yellow, and red LED lights for a traffic light.
- Controlling a fan (attached to an electrical relay) based on data from a humidity sensor.
- Alerting employees in a retail store when customers press a button.
- Using a smart light switch to control other IoT devices.

Note
This connector is not suitable for applications that have real-time requirements. Events with short durations might be missed.

This connector has the following versions.

<table>
<thead>
<tr>
<th>Version</th>
<th>ARN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>arn:aws:greengrass:region::/connectors/RaspberryPiGPIO/versions/1</td>
</tr>
</tbody>
</table>

Requirements

This connector has the following requirements:
- AWS IoT Greengrass Core Software v1.7 or later.
• **Python** version 2.7 installed on the core device and added to the PATH environment variable.

• Raspberry Pi 4 Model B, or Raspberry Pi 3 Model B/B+. You must know the pin sequence of your Raspberry Pi. For more information, see the section called “GPIO Pin Sequence” (p. 366).

• A local device resource (p. 200) in the Greengrass group that points to /dev/gpiomem on the Raspberry Pi. If you create the resource in the console, you must select the **Automatically add OS group permissions of the Linux group that owns the resource** option. In the API, set the **GroupOwnerSetting.AutoAddGroupOwner** property to true.

• The **RPi.GPIO** module installed on the Raspberry Pi. In Raspbian, this module is installed by default. You can use the following command to reinstall it:

```
sudo pip install RPi.GPIO
```

**GPIO Pin Sequence**

The Raspberry Pi GPIO connector references GPIO pins by the numbering scheme of the underlying System on Chip (SoC), not by the physical layout of GPIO pins. The physical ordering of pins might vary in Raspberry Pi versions. For more information, see GPIO in the Raspberry Pi documentation.

The connector can't validate that the input and output pins you configure map correctly to the underlying hardware of your Raspberry Pi. If the pin configuration is invalid, the connector returns a runtime error when it attempts to start on the device. To resolve this issue, reconfigure the connector and then redeploy.

**Note**

Make sure that peripherals for GPIO pins are properly wired to prevent component damage.

**Connector Parameters**

This connector provides the following parameters:

**InputGpios**

A comma-separated list of GPIO pin numbers to configure as inputs. Optionally append `U` to set a pin's pull-up resistor, or `D` to set the pull-down resistor. Example: "5, 6U, 7D".

Display name in console: **Input GPIO pins**

Required: false. You must specify input pins, output pins, or both.

Type: string

Valid pattern: `^\$ | ^[0-9]+[UD]?([0-9]+[UD]?)\*$`

**InputPollPeriod**

The interval (in milliseconds) between each polling operation, which checks input GPIO pins for state changes. The minimum value is 1.

This value depends on your scenario and the type of devices that are polled. For example, a value of 50 should be fast enough to detect a button press.

Display name in console: **Input GPIO polling period**

Required: false

Type: integer

Valid pattern: `^\$ | ^[1-9][0-9]*\$`
OutputGpios

A comma-separated list of GPIO pin numbers to configure as outputs. Optionally append H to set a high state (1), or L to set a low state (0). Example: "8H, 9, 27L".

Display name in console: **Output GPIO pins**

Required: false. You must specify input pins, output pins, or both.

Type: string

Valid pattern: ^\$|^[0-9]+([HL]?)([0-9]+[HL]?)\*$

GpioMem-ResourceId

The ID of the local device resource that represents /dev/gpiomem.

**Note**

This connector is granted read-write access to the resource.

Display name in console: **Resource for /dev/gpiomem device**

Required: true

Type: string

Valid pattern: .+

Create Connector Example (CLI)

The following CLI command creates a ConnectorDefinition with an initial version that contains the Raspberry Pi GPIO connector.

```bash
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-version '{
  "Connectors": [
    {
      "Id": "MyRaspberryPiGPIOConnector",
      "ConnectorArn": "arn:aws:greengrass:region::/connectors/RaspberryPiGPIO/versions/1",
      "Parameters": {
        "GpioMem-ResourceId": "my-gpio-resource",
        "InputGpios": "5,6U,7D",
        "InputPollPeriod": 50,
        "OutputGpios": "8H,9,27L"
      }
    }
  ]
}'
```

**Note**

The Lambda function in this connector has a long-lived (p. 187) lifecycle.

In the AWS IoT Greengrass console, you can add a connector from the group’s Connectors page. For more information, see the section called “Get Started with Connectors (Console)” (p. 403).

Input Data

This connector accepts read or write requests for GPIO pins on two MQTT topics. Input messages must be in JSON format.
• Read requests on the gpio/+/+/read topic.
• Write requests on the gpio/+/+/write topic.

To publish to these topics, replace the + wildcards with the core thing name and the target pin number, respectively. For example:

```
gpio/core-thing-name/gpio-number/read
```

**Note**
Currently, when you create a subscription that uses the Raspberry Pi GPIO connector, you must specify a value for at least one of the + wildcards in the topic.

**Topic filter: gpio/+/+/read**

Use this topic to direct the connector to read the state of the GPIO pin that's specified in the topic.

The connector publishes the response to the corresponding output topic (for example, gpio/core-thing-name/gpio-number/state).

**Message properties**

None. Messages that are sent to this topic are ignored.

**Topic filter: gpio/+/+/write**

Use this topic to send write requests to a GPIO pin. This directs the connector to set the GPIO pin that's specified in the topic to a low or high voltage.

• 0 sets the pin to low voltage.
• 1 sets the pin to high voltage.

The connector publishes the response to the corresponding output /state topic (for example, gpio/core-thing-name/gpio-number/state).

**Message properties**

The value 0 or 1, as an integer or string.

**Example input**

```
0
```

**Output Data**

This connector publishes data to two topics:

• High or low state changes on the gpio/+/+/state topic.
• Errors on the gpio/+/+/error topic.

**Topic filter: gpio/+/+/state**

Use this topic to listen for state changes on input pins and responses for read requests. The connector returns the string "0" if the pin is in a low state, or "1" if it's in a high state.

When publishing to this topic, the connector replaces the + wildcards with the core thing name and the target pin, respectively. For example:

```
gpio/core-thing-name/gpio-number/state
```
Note
Currently, when you create a subscription that uses the Raspberry Pi GPIO connector, you must specify a value for at least one of the + wildcards in the topic.

Example output

| 0 |

**Topic filter:** gpio/+/errors

Use this topic to listen for errors. The connector publishes to this topic as a result of an invalid request (for example, when a state change is requested on an input pin).

When publishing to this topic, the connector replaces the + wildcard with the core thing name.

Example output

```
{
  "topic": "gpio/my-core-thing/22/write",
  "error": "Invalid GPIO operation",
  "long_description": "GPIO 22 is configured as an INPUT GPIO. Write operations are not permitted."
}
```

**Usage Example**

The following example Lambda function sends an input message to the connector. This example sends read requests for a set of input GPIO pins. It shows how to construct topics using the core thing name and pin number.

**Note**
This Python function uses the AWS IoT Greengrass Core SDK (p. 175) to publish an MQTT message. You can use the following pip command to install the Python version of the SDK on your core device:

```
pip install greengrasssdk
```

```python
import greengrasssdk
import json

iot_client = greengrasssdk.client('iot-data')
INPUT_GPIOS = [6, 17, 22]

thingName = os.environ['AWS_IOT_THING_NAME']

def get_read_topic(gpio_num):
    return '/'.join(['gpio', thingName, str(gpio_num), 'read'])

def get_write_topic(gpio_num):
    return '/'.join(['gpio', thingName, str(gpio_num), 'write'])

def send_message_to_connector(topic, message=''):  
    iot_client.publish(topic=topic, payload=str(message))

def set_gpio_state(gpio, state):
    send_message_to_connector(get_write_topic(gpio), str(state))

def read_gpio_state(gpio):
    send_message_to_connector(get_read_topic(gpio))
```
def publish_basic_message():
    for i in INPUT_GPIOS:
        read_gpio_state(i)
publish_basic_message()

def function_handler(event, context):
    return

Licenses

This connector is released under the Greengrass Core Software License Agreement.

See Also

- Integrate with Services and Protocols Using Connectors (p. 283)
- the section called “Get Started with Connectors (Console)” (p. 403)
- the section called “Get Started with Connectors (CLI)” (p. 413)
- GPIO in the Raspberry Pi documentation

Serial Stream Connector

The Serial Stream connector (p. 283) reads and writes to a serial port on an AWS IoT Greengrass core device.

This connector supports two modes of operation:

- **Read-On-Demand.** Receives read and write requests on MQTT topics and publishes the response of the read operation or the status of the write operation.
- **Polling-Read.** Reads from the serial port at regular intervals. This mode also supports Read-On-Demand requests.

**Note**

Read requests are limited to a maximum read length of 63994 bytes. Write requests are limited to a maximum data length of 128000 bytes.

This connector has the following versions.

<table>
<thead>
<tr>
<th>Version</th>
<th>ARN</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>arn:aws:greengrass:region::/connectors/SerialStream/versions/2</td>
</tr>
<tr>
<td>1</td>
<td>arn:aws:greengrass:region::/connectors/SerialStream/versions/1</td>
</tr>
</tbody>
</table>

For information about version changes, see the Changelog (p. 377).

Requirements

This connector has the following requirements:
Serial Stream

- AWS IoT Greengrass Core Software v1.7 or later.
- Python version 2.7 installed on the core device and added to the PATH environment variable.
- A local device resource (p. 200) in the Greengrass group that points to the target serial port.

**Note**
Before you deploy this connector, we recommend that you set up the serial port and verify that it can be read from and written to.

**Connector Parameters**

This connector provides the following parameters:

**BaudRate**

The baud rate of the serial connection.

Display name in console: **Baud rate**

Required: true

Type: string

Valid values: 110, 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 56000, 57600, 115200, 230400

Valid pattern: ^110$|^300$|^600$|^1200$|^2400$|^4800$|^9600$|^14400$|^19200$|^28800$|^38400$|^56000$|^57600$|^115200$|^230400$

**Timeout**

The timeout (in seconds) for a read operation.

Display name in console: **Timeout**

Required: true

Type: string

Valid values: 1 - 59

Valid pattern: ^([1-9]|[1-5][0-9])$

**SerialPort**

The absolute path to the physical serial port on the device. This is the source path that's specified for the local device resource.

Display name in console: **Serial port**

Required: true

Type: string

Valid pattern: [/a-zA-Z0-9_\-]+

**SerialPort-ResourceId**

The ID of the local device resource that represents the physical serial port.

**Note**
This connector is granted read-write access to the resource.
Display name in console: **Serial port resource**

- **Required:** true
- **Type:** string
- **Valid pattern:** `[a-zA-Z0-9-_]+`

### PollingRead

Sets the read mode: Polling-Read or Read-On-Demand.

- For Polling-Read mode, specify `true`. In this mode, the `PollingInterval`, `PollingReadType`, and `PollingReadLength` properties are required.
- For Read-On-Demand mode, specify `false`. In this mode, the type and length values are specified in the read request.

- **Display name in console:** **Read mode**
- **Required:** true
- **Type:** string
- **Valid values:** `true`, `false`
- **Valid pattern:** `^([Tt][Rr][Uu][Ee]|[Ff][Aa][Ll][Ss][Ee])$`

### PollingReadLength

The length of data (in bytes) to read in each polling read operation. This applies only when using Polling-Read mode.

- **Display name in console:** **Polling read length**
- **Required:** false. This property is required when `PollingRead` is `true`.
- **Type:** string
- **Valid values:** `1 - 999`
- **Valid pattern:** `^(\d{1,3})$`

### PollingReadInterval

The interval (in seconds) at which the polling read takes place. This applies only when using Polling-Read mode.

- **Display name in console:** **Polling read interval**
- **Required:** false. This property is required when `PollingRead` is `true`.
- **Type:** string
- **Valid values:** `1 - 999`
- **Valid pattern:** `^(\d{1,3})$`

### PollingReadType

The type of data that the polling thread reads. This applies only when using Polling-Read mode.

- **Display name in console:** **Polling read type**
- **Required:** false. This property is required when `PollingRead` is `true`.
Type: string
Valid values: ascii, hex
Valid pattern: ^([Aa][Ss][Cc][Ii][Ii]|Hh][Ee][Xx])$

RtsCts
Indicates whether to enable the RTS/CTS flow control. The default value is false. For more information, see RTS, CTS, and RTR.

Display name in console: RTS/CTS flow control
Required: false
Type: string
Valid values: true, false
Valid pattern: ^([Tt][Rr][Uu][Ee]|Ff][Aa][Ll][Ss][Ee])$

XonXoff
Indicates whether to enable the software flow control. The default value is false. For more information, see Software flow control.

Display name in console: Software flow control
Required: false
Type: string
Valid values: true, false
Valid pattern: ^([Tt][Rr][Uu][Ee]|Ff][Aa][Ll][Ss][Ee])$

Parity
The parity of the serial port. The default value is N. For more information, see Parity.

Display name in console: Serial port parity
Required: false
Type: string
Valid values: N, E, O, S, M
Valid pattern: ^([NEOSmneosm])$

Create Connector Example (CLI)
The following CLI command creates a ConnectorDefinition with an initial version that contains the Serial Stream connector. It configures the connector for Polling-Read mode.

```
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-version '{
"Connectors": [
  {
    "Id": "MySerialStreamConnector",
```
"ConnectorArn": "arn:aws:greengrass:region::/connectors/SerialStream/versions/2",
"Parameters": {
  "BaudRate": "9600",
  "Timeout": "25",
  "SerialPort": "/dev/serial1",
  "SerialPort-ResourceId": "my-serial-port-resource",
  "PollingRead": "true",
  "PollingReadLength": "30",
  "PollingReadInterval": "30",
  "PollingReadType": "hex"
}
}
"

In the AWS IoT Greengrass console, you can add a connector from the group's Connectors page. For more information, see the section called “Get Started with Connectors (Console)” (p. 403).

Input Data

This connector accepts read or write requests for serial ports on two MQTT topics. Input messages must be in JSON format.

- Read requests on the `serial/+read/#` topic.
- Write requests on the `serial/+write/#` topic.

To publish to these topics, replace the `+` wildcard with the core thing name and `#` wildcard with the path to the serial port. For example:

```
serial/core-thing-name/read/dev/serial-port
```

**Topic filter:** `serial/+read/#`

Use this topic to send on-demand read requests to a serial pin. Read requests are limited to a maximum read length of 63994 bytes.

**Message properties**

- **readLength**
  - The length of data to read from the serial port.
  - Required: true
  - Type: string
  - Valid pattern: `^[1-9][0-9]*$`

- **type**
  - The type of data to read.
  - Required: true
  - Type: string
  - Valid values: ascii, hex
  - Valid pattern: `(?i)^(ascii|hex)$`
id

An arbitrary ID for the request. This property is used to map an input request to an output response.

Required: false
Type: string
Valid pattern: .+

Example input:

```json
{
    "readLength": "30",
    "type": "ascii",
    "id": "abc123"
}
```

**Topic filter:** `serial/+/write/#`

Use this topic to send write requests to a serial pin. Write requests are limited to a maximum data length of 128000 bytes.

**Message properties**

- **data**
  The string to write to the serial port.

  Required: true
  Type: string
  Valid pattern: `^[1-9][0-9]*$`

- **type**
  The type of data to read.

  Required: true
  Type: string
  Valid values: ascii, hex
  Valid pattern: `^(ascii|hex|ASCII|HEX)$`

- **id**
  An arbitrary ID for the request. This property is used to map an input request to an output response.

  Required: false
  Type: string
  Valid pattern: .+

**Example input: ASCII request**

```json
{
    "data": "random serial data",
    "type": "ascii",
    "id": "abc123"
}
```
Example input: hex request

```json
{
    "data": "base64 encoded data",
    "type": "hex",
    "id": "abc123"
}
```

Output Data

The connector publishes output data on two topics:

- Status information from the connector on the `serial/+status/#` topic.
- Responses from read requests on the `serial/+read_response/#` topic.

When publishing to this topic, the connector replaces the `+` wildcard with the core thing name and `#` wildcard with the path to the serial port. For example:

```
serial/core-thing-name/status/dev/serial-port
```

**Topic filter: serial/+status/#**

Use this topic to listen for the status of read and write requests. If an `id` property is included in the request, it's returned in the response.

**Example output: Success**

```json
{
    "response": {
        "status": "success"
    },
    "id": "abc123"
}
```

**Example output: Failure**

A failure response includes an `error_message` property that describes the error or timeout encountered while performing the read or write operation.

```json
{
    "response": {
        "status": "fail",
        "error_message": "Could not write to port"
    },
    "id": "abc123"
}
```

**Topic filter: serial/+read_response/#**

Use this topic to receive response data from a read operation. The response data is Base64 encoded if the type is hex.

**Example output**

```json
{

```
"data": "output of serial read operation"
"id": "abc123"
}

**Usage Example**

The following example Lambda function sends an input message to the connector.

**Note**
This Python function uses the AWS IoT Greengrass Core SDK (p. 175) to publish an MQTT message. You can use the following `pip` command to install the Python version of the SDK on your core device:

```
$ pip install greengrasssdk
```

```python
import greengrasssdk
import json

TOPIC_REQUEST = 'serial/CORE_THING_NAME/write/dev/serial1'

# Creating a greengrass core sdk client
iot_client = greengrasssdk.client('iot-data')

def create_serial_stream_request():
    request = {
        "data": "TEST",
        "type": "ascii",
        "id": "abc123"
    }
    return request

def publish_basic_request():
    iot_client.publish(payload=json.dumps(create_serial_stream_request()),
                      topic=TOPIC_REQUEST)

publish_basic_request()

def function_handler(event, context):
    return
```

**Licenses**

The Serial Stream connector includes the following third-party software/licensing:

- `pyserial`/BSD

This connector is released under the Greengrass Core Software License Agreement.

**Changelog**

The following table describes the changes in each version of the connector.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Updated connector ARN for AWS Region support.</td>
</tr>
</tbody>
</table>
A Greengrass group can contain only one version of the connector at a time.

**See Also**

- Integrate with Services and Protocols Using Connectors (p. 283)
- the section called “Get Started with Connectors (Console)” (p. 403)
- the section called “Get Started with Connectors (CLI)” (p. 413)

**ServiceNow MetricBase Integration Connector**

The ServiceNow MetricBase Integration connector (p. 283) publishes time series metrics from Greengrass devices to ServiceNow MetricBase. This allows you to store, analyze, and visualize time series data from the Greengrass core environment, and act on local events.

This connector receives time series data on an MQTT topic, and publishes the data to the ServiceNow API at regular intervals.

You can use this connector to support scenarios such as:

- Create threshold-based alerts and alarms based on time series data collected from Greengrass devices.
- Use time services data from Greengrass devices with custom applications built on the ServiceNow platform.

This connector has the following versions.

<table>
<thead>
<tr>
<th>Version</th>
<th>ARN</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>arn:aws:greengrass:region::/connectors/ServiceNowMetricBaseIntegration/versions/2</td>
</tr>
<tr>
<td>1</td>
<td>arn:aws:greengrass:region::/connectors/ServiceNowMetricBaseIntegration/versions/1</td>
</tr>
</tbody>
</table>

For information about version changes, see the Changelog (p. 384).

**Requirements**

This connector has the following requirements:

- AWS IoT Greengrass Core Software v1.7 or later. AWS IoT Greengrass must be configured to support local secrets, as described in Secrets Requirements (p. 264).
  
  **Note**
  
  This includes allowing access to your Secrets Manager secrets. If you're using the default Greengrass service role, Greengrass has permission to get the values of secrets with names that start with `greengrass-`.

- Python version 2.7 installed on the core device and added to the PATH environment variable.
• A ServiceNow account with an activated subscription to MetricBase. In addition, a metric and metric table must be created in the account. For more information, see MetricBase in the ServiceNow documentation.

• A text type secret in AWS Secrets Manager that stores the user name and password for your ServiceNow instance (for basic authentication). The secret must contain "user" and "password" keys with corresponding values. For more information, see Creating a Basic Secret in the AWS Secrets Manager User Guide.

• A secret resource in the Greengrass group that references the Secrets Manager secret. For more information, see Deploy Secrets to the Core (p. 263).

Connector Parameters

This connector provides the following parameters:

PublishInterval

The maximum number of seconds to wait between publish events to ServiceNow. The maximum value is 900.

The connector publishes to ServiceNow when PublishBatchSize is reached or PublishInterval expires.

Display name in console: Publish interval in seconds

Required: true

Type: string

Valid values: 1 - 900

Valid pattern: \[1-9]\|[1-9]\d|[1-9]\d\d|900

PublishBatchSize

The maximum number of metric values that can be batched before they are published to ServiceNow.

The connector publishes to ServiceNow when PublishBatchSize is reached or PublishInterval expires.

Display name in console: Publish batch size

Required: true

Type: string

Valid pattern: ^[0-9]+$
DefaultTableName

The name of the table that contains the GlideRecord associated with the time series MetricBase database. The table property in the input message payload can be used to override this value.

Display name in console: **Name of the table to contain the metric**

Required: true
Type: string
Valid pattern: .+

MaxMetricsToRetain

The maximum number of metrics to save in memory before they are replaced with new metrics.

This limit applies when there's no connection to the internet and the connector starts to buffer the metrics to publish later. When the buffer is full, the oldest metrics are replaced by new metrics.

**Note**
Metrics are not saved if the host process for the connector is interrupted. For example, this can happen during group deployment or when the device restarts.

This value should be greater than the batch size and large enough to hold messages based on the incoming rate of the MQTT messages.

Display name in console: **Maximum metrics to retain in memory**

Required: true
Type: string
Valid pattern: ^[0-9]+$

AuthSecretArn

The secret in AWS Secrets Manager that stores the ServiceNow user name and password. This must be a text type secret. The secret must contain "user" and "password" keys with corresponding values.

Display name in console: **ARN of auth secret**

Required: true
Type: string

Valid pattern: arn:aws:secretsmanager:[a-zA-Z0-9\-]+:[0-9]{12}:secret:([a-zA-Z0-9\-]+/)+[a-zA-Z0-9\-]+=,\.[@-]+-[a-zA-Z0-9]+

AuthSecretArn-ResourceId

The secret resource in the group that references the Secrets Manager secret for the ServiceNow credentials.

Display name in console: **Auth token resource**

Required: true
Type: string

Valid pattern: .+
Create Connector Example (CLI)

The following CLI command creates a ConnectorDefinition with an initial version that contains the ServiceNow MetricBase Integration connector.

```bash
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-version
'{
   "Connectors": [
    {
       "Id": "MyServiceNowMetricBaseIntegrationConnector",
       "ConnectorArn": "arn:aws:greengrass:region::/connectors/ServiceNowMetricBaseIntegration/versions/2",
       "Parameters": {
           "PublishInterval": "10",
           "PublishBatchSize": "50",
           "InstanceName": "myinstance",
           "DefaultTableName": "u_greengrass_app",
           "MaxMetricsToRetain": "20000",
           "AuthSecretArn-ResourceId": "MySecretResource"
       }
    }
   ]
'}
```

**Note**
The Lambda function in this connector has a long-lived (p. 187) lifecycle.

In the AWS IoT Greengrass console, you can add a connector from the group's Connectors page. For more information, see the section called "Get Started with Connectors (Console)" (p. 403).

**Input Data**

This connector accepts time series metrics on an MQTT topic and publishes the metrics to ServiceNow. Input messages must be in JSON format.

**Topic filter**
```
servicenow/metricbase/metric
```

**Message properties**
- **request**
  
  Information about the table, record, and metric. This request represents the seriesRef object in a time series POST request. For more information, see Clotho Time Series API - POST.

  **Required**: true

  **Type**: object that includes the following properties:

  - **subject**
    
    The sys_id of the specific record in the table.

    **Required**: true

    **Type**: string

  - **metric_name**
    
    The metric field name.
Required: true
Type: string
table
The name of the table to store the record in. Specify this value to override the DefaultTableName parameter.
Required: false
Type: string
value
The value of the individual data point.
Required: true
Type: float
timestamp
The timestamp of the individual data point. The default value is the current time.
Required: false
Type: string

Example input

```json
{
  "request": {
    "subject": "ef43c6d40a0a0b5700c77f9bf387afe3",
    "metric_name": "u_count",
    "table": "u_greengrass_app",
    "value": 1.0,
    "timestamp": "2018-10-14T10:30:00"
  }
}
```

Output Data

This connector publishes status information as output data.

**Topic filter**

`servicenow/metricbase/metric/status`

**Example output: Success**

```json
{
  "response": {
    "metric_name": "Errors",
    "table_name": "GliderProd",
    "processed_on": "2018-10-14T10:35:00",
    "response_id": "khjKSkj132qwr23fcba",
    "status": "success",
    "values": [
      {
        "timestamp": "2016-10-14T10:30:00",
        "value": 1.0
      }
    ]
  }
}
```
Example output: Failure

```
{
  "response": {
    "error": "InvalidInputException",
    "error_message": "metric value is invalid",
    "status": "fail"
  }
}
```

**Note**
If the connector detects a retryable error (for example, connection errors), it retries the publish in the next batch.

**Usage Example**

The following example Lambda function sends an input message to the connector.

**Note**
This Python function uses the [AWS IoT Greengrass Core SDK (p. 175)](https://docs.aws.amazon.com/greengrass/latest/developer-guide/) to publish an MQTT message. You can use the following `pip` command to install the Python version of the SDK on your core device:

```
pip install greengrasssdk
```

```python
import greengrasssdk
import json

iot_client = greengrasssdk.client('iot-data')
SEND_TOPIC = 'servicenow/metricbase/metric'

def create_request_with_all_fields():
    return {
        "request": {
            "subject": '2efdf6badbd523803acfae441b961961',
            "metric_name": 'u_count',
            "value": 1234,
            "timestamp": '2018-10-20T20:22:20',
            "table": 'u_greengrass_metricbase_test'
        }
    }

def publish_basic_message():
    messageToPublish = create_request_with_all_fields()
    print "Message To Publish: ", messageToPublish
    iot_client.publish(topic=SEND_TOPIC,
                      payload=json.dumps(messageToPublish))

    publish_basic_message()

def function_handler(event, context):
    return
```
Licenses

The ServiceNow MetricBase Integration connector includes the following third-party software/licensing:

- pysnow/MIT

This connector is released under the Greengrass Core Software License Agreement.

Changelog

The following table describes the changes in each version of the connector.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Fix to reduce excessive logging.</td>
</tr>
<tr>
<td>1</td>
<td>Initial release.</td>
</tr>
</tbody>
</table>

A Greengrass group can contain only one version of the connector at a time.

See Also

- Integrate with Services and Protocols Using Connectors (p. 283)
- the section called “Get Started with Connectors (Console)” (p. 403)
- the section called “Get Started with Connectors (CLI)” (p. 413)

SNS Connector

The SNS connector (p. 283) publishes messages to an Amazon SNS topic. This enables web servers, email addresses, and other message subscribers to respond to events in the Greengrass group.

This connector receives SNS message information on an MQTT topic, and then sends the message to a specified SNS topic. You can optionally use custom Lambda functions to implement filtering or formatting logic on messages before they are published to this connector.

This connector has the following versions.

<table>
<thead>
<tr>
<th>Version</th>
<th>ARN</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>arn:aws:greengrass:region::/connectors/SNS/versions/2</td>
</tr>
<tr>
<td>1</td>
<td>arn:aws:greengrass:region::/connectors/SNS/versions/1</td>
</tr>
</tbody>
</table>

For information about version changes, see the Changelog (p. 389).

Requirements

This connector has the following requirements:
• AWS IoT Greengrass Core Software v1.7 or later.
• Python version 2.7 installed on the core device and added to the PATH environment variable.
• A configured SNS topic. For more information, see Creating an Amazon SNS Topic in the Amazon Simple Notification Service Developer Guide.
• An IAM policy added to the Greengrass group role that allows the sns:Publish action on the target SNS topic, as shown in the following example:

```json
{
  "Version":"2012-10-17",
  "Statement":[
    {
      "Sid":"Stmt1528133056761",
      "Action":[
        "sns:Publish"
      ],
      "Effect":"Allow",
      "Resource":[
        "arn:aws:sns:region:account-id:topic-name"
      ]
    }
  ]
}
```

This connector allows you to dynamically override the default topic in the input message payload. If your implementation uses this feature, the IAM policy must allow sns:Publish permission on all target topics. 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.

**Connector Parameters**

This connector provides the following parameters:

**DefaultSNSArn**

The ARN of the default SNS topic to publish messages to. The destination topic can be overridden by the sns_topic_arn property in the input message payload.

**Note**

The group role must allow sns:Publish permission to all target topics. For more information, see the section called “Requirements” (p. 384).

Display name in console: **Default SNS topic ARN**

Required: true

Type: string

Valid pattern: arn:aws:sns:([a-z]{2}-[a-z]+-\d{1}):\d{12}:([a-zA-Z0-9-\_]+)*

**Create Connector Example (CLI)**

The following CLI command creates a ConnectorDefinition with an initial version that contains the SNS connector.

```
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-version '{
```
In the AWS IoT Greengrass console, you can add a connector from the group’s Connectors page. For more information, see the section called “Get Started with Connectors (Console)” (p. 403).

### Input Data

This connector accepts SNS message information on an MQTT topic, and then publishes the message as is to the target SNS topic. Input messages must be in JSON format.

#### Topic filter

```
sns/message
```

#### Message properties

**request**

Information about the message to send to the SNS topic.

Required: true

Type: object that includes the following properties:

- **message**

  The content of the message as a string or in JSON format. For examples, see Example input (p. 387).

  To send JSON, the message_structure property must be set to json and the message must be a string-encoded JSON object that contains a default key.

  Required: true

  Type: string

  Valid pattern: .*

- **subject**

  The subject of the message.

  Required: false

  Type: ASCII text, up to 100 characters. This must begin with a letter, number, or punctuation mark. This must not include line breaks or control characters.

  Valid pattern: .*

- **sns_topic_arn**

  The ARN of the SNS topic to publish messages to. If specified, the connector publishes to this topic instead of the default topic.
Note
The group role must allow sns:Publish permission to any target topics. For more information, see the section called “Requirements” (p. 384).

Required: false
Type: string
Valid pattern: arn:aws:sns:([a-z]{2}-[a-z]+-[a-z0-9-]+):([a-zA-20-9-]+)$

message_structure
The structure of the message.
Required: false. This must be specified to send a JSON message.
Type: string
Valid values: json

id
An arbitrary ID for the request. This property is used to map an input request to an output response. When specified, the id property in the response object is set to this value. If you don’t use this feature, you can omit this property or specify an empty string.
Required: false
Type: string
Valid pattern: .*

Limits
The message size is bounded by a maximum SNS message size of 256 KB.

Example input: String message
This example sends a string message. It specifies the optional sns_topic_arn property, which overrides the default destination topic.

```
{
   "request": {
      "subject": "Message subject",
      "message": "Message data",
      "sns_topic_arn": "arn:aws:sns:region:account-id:topic2-name"
   },
   "id": "request123"
}
```

Example input: JSON message
This example sends a message as a string encoded JSON object that includes the default key.

```
{
   "request": {
      "subject": "Message subject",
      "message": "{ \"default\": \"Message data\" }",
      "message_structure": "json"
   },
   "id": "request123"
}
```
Output Data

This connector publishes status information as output data.

Topic filter

sns/message/status

Example output: Success

```json
{
    "response": {
        "sns_message_id": "f80a81bc-f44c-56f2-a0f0-d5af6a727c8a",
        "status": "success"
    },
    "id": "request123"
}
```

Example output: Failure

```json
{
    "response": {
        "error": "InvalidInputException",
        "error_message": "SNS Topic Arn is invalid",
        "status": "fail"
    },
    "id": "request123"
}
```

Usage Example

The following example Lambda function sends an input message to the connector.

**Note**

This Python function uses the AWS IoT Greengrass Core SDK (p. 175) to publish an MQTT message. You can use the following pip command to install the Python version of the SDK on your core device:

```
pip install greengrasssdk
```

```python
import greengrasssdk
import time
import json

iot_client = greengrasssdk.client('iot-data')
send_topic = 'sns/message'

def create_request_with_all_fields():
    return {
        "request": {
            "message": "Message from SNS Connector Test"
        },
        "id": "req_123"
    }

def publish_basic_message():
    messageToPublish = create_request_with_all_fields()
    print "Message To Publish: ", messageToPublish
    iot_client.publish(topic=send_topic,
                       payload=json.dumps(messageToPublish))
```

388
Licenses

The SNS connector includes the following third-party software/licensing:

- AWS SDK for Python (Boto 3)/Apache 2.0

This connector is released under the Greengrass Core Software License Agreement.

Changelog

The following table describes the changes in each version of the connector.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Fix to reduce excessive logging.</td>
</tr>
<tr>
<td>1</td>
<td>Initial release.</td>
</tr>
</tbody>
</table>

A Greengrass group can contain only one version of the connector at a time.

See Also

- Integrate with Services and Protocols Using Connectors (p. 283)
- the section called “Get Started with Connectors (Console)” (p. 403)
- the section called “Get Started with Connectors (CLI)” (p. 413)
- Publish action in the Boto 3 documentation
- What Is Amazon Simple Notification Service? in the Amazon Simple Notification Service Developer Guide

Splunk Integration Connector

The Splunk Integration connector (p. 283) publishes data from Greengrass devices to Splunk. This allows you to use Splunk to monitor and analyze the Greengrass core environment, and act on local events. The connector integrates with HTTP Event Collector (HEC). For more information, see Introduction to Splunk HTTP Event Collector in the Splunk documentation.

This connector receives logging and event data on an MQTT topic and publishes the data as is to the Splunk API.

You can use this connector to support industrial scenarios, such as:

- Operators can use periodic data from actuators and sensors (for example, temperature, pressure, and water readings) to trigger alarms when values exceed certain thresholds.
- Developers use data collected from industrial machinery to build ML models that can monitor the equipment for potential issues.

This connector has the following versions.
Version | ARN
--- | ---
2 | arn:aws:greengrass:region::/connectors/SplunkIntegration/versions/2
1 | arn:aws:greengrass:region::/connectors/SplunkIntegration/versions/1

For information about version changes, see the Changelog (p. 394).

Requirements

This connector has the following requirements:

- AWS IoT Greengrass Core Software v1.7 or later. AWS IoT Greengrass must be configured to support local secrets, as described in Secrets Requirements (p. 264).
  
  **Note**
  This includes allowing access to your Secrets Manager secrets. If you're using the default Greengrass service role, Greengrass has permission to get the values of secrets with names that start with `greengrass-`.

- Python version 2.7 installed on the core device and added to the PATH environment variable.

- The HTTP Event Collector functionality must be enabled in Splunk. For more information, see Set up and use HTTP Event Collector in Splunk Web in the Splunk documentation.

- A text type secret in AWS Secrets Manager that stores your Splunk HTTP Event Collector token. For more information, see About Event Collector tokens in the Splunk documentation and Creating a Basic Secret in the AWS Secrets Manager User Guide.
  
  **Note**
  To create the secret in the Secrets Manager console, enter your token on the Plaintext tab. Don't include quotation marks or other formatting. In the API, specify the token as the value for the SecretString property.

- A secret resource in the Greengrass group that references the Secrets Manager secret. For more information, see Deploy Secrets to the Core (p. 263).

Connector Parameters

This connector provides the following parameters:

**SplunkEndpoint**

The endpoint of your Splunk instance. This value must contain the protocol, hostname, and port.

Display name in console: **Splunk endpoint**

Required: true

Type: string

Valid pattern: `^http:\/\/[a-z0-9]+([-\.]\{1\}[a-z0-9]+\*\.[a-z]{2,5}(:[0-9]\{1,5\})?(\/*))?\$`

**MemorySize**

The amount of memory (in KB) to allocate to the connector.

Display name in console: **Memory size**
Required: true
Type: string
Valid pattern: ^[0-9]+$  
SplunkQueueSize
The maximum number of items to save in memory before the items are submitted or discarded. When this limit is met, the oldest items in the queue are replaced with newer items. This limit typically applies when there's no connection to the internet.
Display name in console: **Maximum items to retain**
Required: true
Type: string
Valid pattern: ^[0-9]+$  
SplunkFlushIntervalSeconds
The interval (in seconds) for publishing received data to Splunk HEC. The maximum value is 900. To configure the connector to publish items as they are received (without batching), specify 0.
Display name in console: **Splunk publish interval**
Required: true
Type: string
Valid pattern: [0-9][1-9]\d[1-9]\d\d|900  
SplunkTokenSecretArn
The secret in AWS Secrets Manager that stores the Splunk token. This must be a text type secret.
Display name in console: **ARN of Splunk auth token secret**
Required: true
Type: string
Valid pattern: arn:aws:secretsmanager:[a-z]{2}-[a-z]+-\d{1}:\d{12}?:\?secret:\[a-zA-Z0-9_-]+[a-zA-Z0-9-_.]+  
SplunkTokenSecretArn-ResourceId
The secret resource in the Greengrass group that references the Splunk secret.
Display name in console: **Splunk auth token resource**
Required: true
Type: string
Valid pattern: .+  
SplunkCustomCALocation
The file path of the custom certificate authority (CA) for Splunk (for example, /etc/ssl/certs/splunk.crt).
Display name in console: **Splunk custom certificate authority location**
Required: false
Create Connector Example (CLI)

The following CLI command creates a ConnectorDefinition with an initial version that contains the Splunk Integration connector.

```bash
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-version '{
  "Connectors": [
    {
      "Id": "MySplunkIntegrationConnector",
      "ConnectorArn": "arn:aws:greengrass:region::/connectors/SplunkIntegration/versions/2",
      "Parameters": {
        "SplunkEndpoint": "https://myinstance.cloud.splunk.com:8088",
        "MemorySize": 200000,
        "SplunkQueueSize": 10000,
        "SplunkFlushIntervalSeconds": 5,
        "SplunkTokenSecretArn-ResourceId": "MySplunkResource"
      }
    }
  ]
}
```

Note

The Lambda function in this connector has a long-lived (p. 187) lifecycle.

In the AWS IoT Greengrass console, you can add a connector from the group's Connectors page. For more information, see the section called “Get Started with Connectors (Console)” (p. 403).

Input Data

This connector accepts logging and event data on an MQTT topic and publishes the received data as is to the Splunk API. Input messages must be in JSON format.

**Topic filter**

```
splunk/logs/put
```

**Message properties**

`request`

The event data to send to the Splunk API. Events must meet the specifications of the services/collector API.

Required: true

Type: object. Only the event property is required.

`id`

An arbitrary ID for the request. This property is used to map an input request to an output status.

Required: false
Type: string

Limits

All limits that are imposed by the Splunk API apply when using this connector. For more information, see services/collector.

Example input

```json
{
  "request": {
    "event": "some event",
    "fields": {
      "severity": "INFO",
      "category": [
        "value1",
        "value2"
      ]
    }
  },
  "id": "request123"
}
```

Output Data

This connector publishes output data on two topics:

- Status information on the splunk/logs/put/status topic.
- Errors on the splunk/logs/put/error topic.

**Topic filter: splunk/logs/put/status**

Use this topic to listen for the status of the requests. Each time that the connector sends a batch of received data to the Splunk API, it publishes a list of the IDs of the requests that succeeded and failed.

**Example output**

```json
{
  "response": {
    "succeeded": [
      "request123",
      ...
    ],
    "failed": [
      "request789",
      ...
    ]
  }
}
```

**Topic filter: splunk/logs/put/error**

Use this topic to listen for errors from the connector. The error_message property that describes the error or timeout encountered while processing the request.

**Example output**

```json
{
  "response": {
    "error": "UnauthorizedException",
    "message": "request failed with UnauthorizedException"
  }
}
```
"error_message": "invalid splunk token",
"status": "fail"
}

**Note**
If the connector detects a retryable error (for example, connection errors), it retries the publish in the next batch.

### Usage Example

The following example Lambda function sends an input message to the connector.

**Note**
This Python function uses the AWS IoT Greengrass Core SDK (p. 175) to publish an MQTT message. You can use the following pip command to install the Python version of the SDK on your core device:

```
pip install greengrasssdk
```

```python
import greengrasssdk
import time
import json

iot_client = greengrasssdk.client('iot-data')
send_topic = 'splunk/logs/put'

def create_request_with_all_fields():
    return {
        "request": {
            "event": "Access log test message."
        },
        "id": "req_123"
    }

def publish_basic_message():
    messageToPublish = create_request_with_all_fields()
    print "Message To Publish: ", messageToPublish
    iot_client.publish(topic=send_topic,
                       payload=json.dumps(messageToPublish))

publish_basic_message()

def function_handler(event, context):
    return
```

### Licenses

This connector is released under the Greengrass Core Software License Agreement.

### Changelog

The following table describes the changes in each version of the connector.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Fix to reduce excessive logging.</td>
</tr>
</tbody>
</table>
A Greengrass group can contain only one version of the connector at a time.

**See Also**

- *Integrate with Services and Protocols Using Connectors* (p. 283)
- the section called “Get Started with Connectors (Console)” (p. 403)
- the section called “Get Started with Connectors (CLI)” (p. 413)

**Twilio Notifications Connector**

The Twilio Notifications connector (p. 283) makes automated phone calls or sends text messages through Twilio. You can use this connector to send notifications in response to events in the Greengrass group. For phone calls, the connector can forward a voice message to the recipient.

This connector receives Twilio message information on an MQTT topic, and then triggers a Twilio notification.

**Note**

For a tutorial that shows how to use the Twilio Notifications connector, see the section called “Get Started with Connectors (Console)” (p. 403) or the section called “Get Started with Connectors (CLI)” (p. 413).

This connector has the following versions.

<table>
<thead>
<tr>
<th>Version</th>
<th>ARN</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>arn:aws:greengrass:region::/connectors/TwilioNotifications/versions/3</td>
</tr>
<tr>
<td>2</td>
<td>arn:aws:greengrass:region::/connectors/TwilioNotifications/versions/2</td>
</tr>
<tr>
<td>1</td>
<td>arn:aws:greengrass:region::/connectors/TwilioNotifications/versions/1</td>
</tr>
</tbody>
</table>

For information about version changes, see the Changelog (p. 402).

**Requirements**

This connector has the following requirements:

- AWS IoT Greengrass Core Software v1.7 or later. AWS IoT Greengrass must be configured to support local secrets, as described in Secrets Requirements (p. 264).

 **Note**

This includes allowing access to your Secrets Manager secrets. If you're using the default Greengrass service role, Greengrass has permission to get the values of secrets with names that start with `greengrass-`.

- Python version 2.7 installed on the core device and added to the PATH environment variable.
• A Twilio account SID, auth token, and Twilio-enabled phone number. After you create a Twilio project, these values are available on the project dashboard.

  **Note**
  You can use a Twilio trial account. If you’re using a trial account, you must add non-Twilio recipient phone numbers to a list of verified phone numbers. For more information, see [How to Work with your Free Twilio Trial Account](#).

• A text type secret in AWS Secrets Manager that stores the Twilio auth token. For more information, see [Creating a Basic Secret](#) in the AWS Secrets Manager User Guide.

  **Note**
  To create the secret in the Secrets Manager console, enter your token on the **Plaintext** tab. Don’t include quotation marks or other formatting. In the API, specify the token as the value for the **SecretString** property.

• A secret resource in the Greengrass group that references the Secrets Manager secret. For more information, see [Deploy Secrets to the Core](#) (p. 263).

### Connector Parameters

This connector provides the following parameters.

**TWILIO_ACCOUNT_SID**

The Twilio account SID that's used to invoke the Twilio API.

- Display name in console: **Twilio account SID**
- Required: `true`
- Type: string
- Valid pattern: `.+`

**TwilioAuthTokenSecretArn**

The ARN of the Secrets Manager secret that stores the Twilio auth token.

  **Note**
  This is used to access the value of the local secret on the core.

- Display name in console: **ARN of Twilio auth token secret**
- Required: `true`
- Type: string

**TwilioAuthTokenSecretArn-ResourceId**

The ID of the secret resource in the Greengrass group that references the secret for the Twilio auth token.

- Display name in console: **Twilio auth token resource**
- Required: `true`
- Type: string
Valid pattern: .+

DefaultFromPhoneNumber

The default Twilio-enabled phone number that Twilio uses to send messages. Twilio uses this number to initiate the text or call.

- If you don't configure a default phone number, you must specify a phone number in the `from_number` property in the input message body.
- If you do configure a default phone number, you can optionally override the default by specifying the `from_number` property in the input message body.

Display name in console: **Default from phone number**

Required: false

Type: string

Valid pattern: ^$|\+[0-9]+

### Create Connector Example (CLI)

The following example CLI command creates a `ConnectorDefinition` with an initial version that contains the Twilio Notifications connector.

```bash
aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-version '{
  "Connectors": [
    {
      "Id": "MyTwilioNotificationsConnector",
      "ConnectorArn": "arn:aws:greengrass:region::/connectors/TwilioNotifications/
        versions/3",
      "Parameters": {
        "TWILIO_ACCOUNT_SID": "abcd12345xyz",
        "TwilioAuthTokenSecretArn-ResourceId": "MyTwilioSecret",
        "DefaultFromPhoneNumber": "+19999999999"
      }
    }
  ],
}
''
```

For tutorials that show how add the Twilio Notifications connector to a group, see the section called “Get Started with Connectors (CLI)” (p. 413) and the section called “Get Started with Connectors (Console)” (p. 403).

### Input Data

This connector accepts Twilio message information on two MQTT topics. Input messages must be in JSON format.

- Text message information on the `twilio/txt` topic.
- Phone message information on the `twilio/call` topic.

**Note**

The input message payload can include a text message (`message`) or voice message (`voice_message_location`), but not both.
### Topic filter: twilio/txt

#### Message properties

**request**

Information about the Twilio notification.

Required: true

Type: object that includes the following properties:

**recipient**

The message recipient. Only one recipient is supported.

Required: true

Type: object that include the following properties:

**name**

The name of the recipient.

Required: true

Type: string

Valid pattern: .*

**phone_number**

The phone number of the recipient.

Required: true

Type: string

Valid pattern: \+[1-9]+ 

**message**

The text content of the text message. Only text messages are supported on this topic. For voice messages, use twilio/call.

Required: true

Type: string

Valid pattern: .+

**from_number**

The phone number of the sender. Twilio uses this phone number to initiate the message. This property is required if the DefaultFromPhoneNumber parameter isn’t configured. If DefaultFromPhoneNumber is configured, you can use this property to override the default.

Required: false

Type: string

Valid pattern: \+[1-9]+ 

**retries**

The number of retries. The default is 0.
Required: false
Type: integer

id

An arbitrary ID for the request. This property is used to map an input request to an output response.

Required: true
Type: string
Valid pattern: .+

Example input

```
{
  "request": {
    "recipient": {
      "name": "Darla",
      "phone_number": "+12345000000",
      "message": "Hello from the edge"
    },
    "from_number": "+19999999999",
    "retries": 3
  },
  "id": "request123"
}
```

Topic filter: twilio/call

Message properties

request

Information about the Twilio notification.

Required: true
Type: object that includes the following properties:

recipient

The message recipient. Only one recipient is supported.

Required: true
Type: object that include the following properties:

name

The name of the recipient.

Required: true
Type: string
Valid pattern: .+

phone_number

The phone number of the recipient.

Required: true
Type: string

Valid pattern: \+[1-9]+

voice_message_location

The URL of the audio content for the voice message. This must be in TwiML format. Only voice messages are supported on this topic. For text messages, use twilio/txt.

Required: true

Type: string

Valid pattern: .+

from_number

The phone number of the sender. Twilio uses this phone number to initiate the message. This property is required if the DefaultFromPhoneNumber parameter isn't configured. If DefaultFromPhoneNumber is configured, you can use this property to override the default.

Required: false

Type: string

Valid pattern: \+[1-9]+

retries

The number of retries. The default is 0.

Required: false

Type: integer

id

An arbitrary ID for the request. This property is used to map an input request to an output response.

Required: true

Type: string

Valid pattern: .+

Example input

```json
{
  "request": {
    "recipient": {
      "name": "Darla",
      "phone_number": "+12345000000",
      "voice_message_location": "https://some-public-TwiML"
    },
    "from_number": "+19999999999",
    "retries": 3
  },
  "id": "request123"
}
```
Output Data

This connector publishes status information as output data.

**Topic filter**

twilio/message/status

**Example output: Success**

```
{
    "response": {
        "status": "success",
        "payload": {
            "from_number": "+19999999999",
            "messages": {
                "message_status": "queued",
                "to_number": "+12345000000",
                "name": "Darla"
            }
        }
    },
    "id": "request123"
}
```

**Example output: Failure**

```
{
    "response": {
        "status": "fail",
        "error_message": "Recipient name cannot be None",
        "error": "InvalidParameter",
        "payload": None
    },
    "id": "request123"
}
```

The **payload** property in the output is the response from the Twilio API when the message is sent. If the connector detects that the input data is invalid (for example, it doesn't specify a required input field), the connector returns an error and sets the value to **None**. The following are example payloads:

```
{
    'from_number': '+19999999999',
    'messages': {
        'name': 'Darla',
        'to_number': '+12345000000',
        'message_status': 'undelivered'
    }
}
```

```
{
    'from_number': '+19999999999',
    'messages': {
        'name': 'Darla',
        'to_number': '+12345000000',
        'message_status': 'queued'
    }
}
```
Usage Example

The following example Lambda function sends an input message to the connector. This example triggers a text message.

**Note**

This Python function uses the AWS IoT Greengrass Core SDK (p. 175) to publish an MQTT message. You can use the following `pip` command to install the Python version of the SDK on your core device:

```
pip install greengrasssdk
```

```python
import greengrasssdk
import json

iot_client = greengrasssdk.client('iot-data')
TXT_INPUT_TOPIC = 'twilio/txt'
CALL_INPUT_TOPIC = 'twilio/call'

def publish_basic_message():
    txt = {
        "request": {
            "recipient" : {
                "name": "Darla",
                "phone_number": "+12345000000",
                "message": 'Hello from the edge'
            },
            "from_number" : "+19999999999"
        },
        "id" : "request123"
    }
    print "Message To Publish: ", txt
    client.publish(topic=TXT_INPUT_TOPIC, payload=json.dumps(txt))

publish_basic_message()

def function_handler(event, context):
    return
```

Licenses

The Twilio Notifications connector includes the following third-party software/licensing:

- twilio-python/MIT

This connector is released under the Greengrass Core Software License Agreement.

Changelog

The following table describes the changes in each version of the connector.

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Fix to reduce excessive logging.</td>
</tr>
</tbody>
</table>
A Greengrass group can contain only one version of the connector at a time.

See Also

• *Integrate with Services and Protocols Using Connectors (p. 283)*
• the section called “Get Started with Connectors (Console)” (p. 403)
• the section called “Get Started with Connectors (CLI)” (p. 413)
• Twilio API Reference

Getting Started with Greengrass Connectors (Console)

This feature is available for AWS IoT Greengrass Core v1.7 and later.

This tutorial shows how to use the AWS Management Console to work with connectors.

Use connectors to accelerate your development life cycle. Connectors are prebuilt, reusable modules that can make it easier to interact with services, protocols, and resources. They can help you deploy business logic to Greengrass devices more quickly. For more information, see *Integrate with Services and Protocols Using Connectors (p. 283)*.

In this tutorial, you configure and deploy the Twilio Notifications (p. 395) connector. The connector receives Twilio message information as input data, and then triggers a Twilio text message. The data flow is shown in following diagram.

After you configure the connector, you create a Lambda function and a subscription.

• The function evaluates simulated data from a temperature sensor. It conditionally publishes the Twilio message information to an MQTT topic. This is the topic that the connector subscribes to.
• The subscription allows the function to publish to the topic and the connector to receive data from the topic.
The Twilio Notifications connector requires a Twilio auth token to interact with the Twilio API. The token is a text type secret created in AWS Secrets Manager and referenced from a group resource. This enables AWS IoT Greengrass to create a local copy of the secret on the Greengrass core, where it is encrypted and made available to the connector. For more information, see Deploy Secrets to the Core (p. 263).

The tutorial contains the following high-level steps:

1. Create a Secrets Manager Secret (p. 404)
2. Add a Secret Resource to a Group (p. 405)
3. Add a Connector to the Group (p. 406)
4. Create a Lambda Function Deployment Package (p. 407)
5. Create a Lambda Function (p. 408)
6. Add a Function to the Group (p. 409)
7. Add Subscriptions to the Group (p. 410)
8. Deploy the Group (p. 411)

The tutorial should take about 20 minutes to complete.

Prerequisites

To complete this tutorial, you need:

- A Greengrass group and a Greengrass core (v1.7 or later). To learn how to create a Greengrass group and core, see Getting Started with AWS IoT Greengrass (p. 64). The Getting Started tutorial also includes steps for installing the AWS IoT Greengrass Core software.
- AWS IoT Greengrass must be configured to support local secrets, as described in Secrets Requirements (p. 264).
  
  **Note**
  This includes allowing access to your Secrets Manager secrets. If you're using the default Greengrass service role, Greengrass has permission to get the values of secrets with names that start with `greengrass-`.

- A Twilio account SID, auth token, and Twilio-enabled phone number. After you create a Twilio project, these values are available on the project dashboard.
  
  **Note**
  You can use a Twilio trial account. If you're using a trial account, you must add non-Twilio recipient phone numbers to a list of verified phone numbers. For more information, see How to Work with your Free Twilio Trial Account.

Step 1: Create a Secrets Manager Secret

In this step, you use the AWS Secrets Manager console to create a text type secret for your Twilio auth token.

1. Sign in to the AWS Secrets Manager console.

   **Note**
   For more information about this process, see Step 1: Create and Store Your Secret in AWS Secrets Manager in the AWS Secrets Manager User Guide.

2. Choose Store a new secret.
3. Under Select secret type, choose Other type of secrets.
4. Under Specify the key/value pairs to be stored for this secret, on the Plaintext tab, enter your Twilio auth token. Remove all of the JSON formatting and enter only the token value.
5. Keep DefaultEncryptionKey selected for the encryption key, and then choose Next.

   Note
   You aren’t charged by AWS KMS if you use the default AWS managed key that Secrets
   Manager creates in your account.

6. For Secret name, enter greengrass-TwilioAuthToken, and then choose Next.

   Note
   By default, the Greengrass service role allows AWS IoT Greengrass to get the value
   of secrets with names that start with greengrass-. For more information, see secrets
   requirements (p. 264).

7. This tutorial doesn’t require rotation, so choose Disable automatic rotation, and then choose Next.

8. On the Review page, review your settings, and then choose Store.

Next, you create a secret resource in your Greengrass group that references the secret.

**Step 2: Add a Secret Resource to a Greengrass Group**

In this step, you add a secret resource to the Greengrass group. This resource is a reference to the secret
that you created in the previous step.

1. In the AWS IoT console, choose Greengrass, and then choose Groups.
2. Choose the group that you want to add the secret resource to.
3. On the group configuration page, choose Resources, and then choose Secret. This tab displays the
secret resources that belong to the group. You can add, edit, and remove secret resources from this tab.
Add a Connector to the Group

Step 3: Add a Connector to the Greengrass Group

In this step, you configure parameters for the Twilio Notifications connector (p. 395) and add it to the group.

1. On the group configuration page, choose Connectors, and then choose Add a connector.

Note
Alternatively, the console allows you to create a secret and secret resource when you configure a connector or Lambda function. You can do this from the connector’s Configure parameters page or the Lambda function’s Resources page.

4. Choose Add a secret resource.

5. On the Add a secret resource to your group page, choose Select, and then choose greengrass-TwilioAuthToken.

6. On the Select labels (Optional) page, choose Next. The AWSCURRENT staging label represents the latest version of the secret. This label is always included in a secret resource.

   Note
   This tutorial requires the AWSCURRENT label only. You can optionally include labels that are required by your Lambda function or connector.

7. On the Name your secret resource page, enter MyTwilioAuthToken, and then choose Save.
2. On the Select a connector page, choose Twilio Notifications, and then choose Next.

3. On the Configure parameters page:
   - For Twilio auth token resource, choose MyTwilioAuthToken. This is the secret resource that you created in the previous step.
     **Note**
     When you choose the resource, the ARN of Twilio auth token secret property is populated for you.
   - For Default from phone number, enter your Twilio-enabled phone number.
   - For Twilio account SID, enter your Twilio account SID.

4. Choose Add.

**Step 4: Create a Lambda Function Deployment Package**

To create a Lambda function, you must first create a Lambda function deployment package that contains the function code and dependencies. Greengrass Lambda functions require the AWS IoT Greengrass Core SDK (p. 175) for tasks such as communicating with MQTT messages in the core environment and accessing local secrets. This tutorial creates a Python function, so you use the Python version of the SDK in the deployment package.

1. Download the AWS IoT Greengrass Core SDK for Python from the AWS IoT Greengrass Core SDK (p. 22) downloads page.
2. Unzip the downloaded package to get the SDK. The SDK is the greengrasssdk folder.
3. Save the following Python code function in a local file named temp_monitor.py.

```python
from __future__ import print_function
import greengrasssdk
import json
import random

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

# publish to the Twilio Notifications connector through the twilio/txt topic
def function_handler(event, context):
    temp = event['temperature']

    # check the temperature
    if temp > 30:
```
Create a Lambda Function

```python
data = build_request(event)
client.publish(topic='twilio/txt', payload=json.dumps(data))
print('published:' + str(data))
print('temperature:' + str(temp))
return

# build the Twilio request from the input data
def build_request(event):
to_name = event['to_name']
to_number = event['to_number']
temp_report = 'temperature:' + str(event['temperature'])
return {
    "request": {
        "recipient": {
            "name": to_name,
            "phone_number": to_number,
            "message": temp_report
        }
    },
    "id": "request_" + str(random.randint(1,101))
}
```

4. Zip the following items into a file named `temp_monitor_python.zip`. When creating the ZIP file, include only the code and dependencies, not the containing folder.

- `temp_monitor.py`. App logic.
- `greengrasssdk`. Required library for Python Greengrass Lambda functions that publish MQTT messages.

This is your Lambda function deployment package.

Now, create a Lambda function that uses the deployment package.

**Step 5: Create a Lambda Function in the AWS Lambda Console**

In this step, you use the AWS Lambda console to create a Lambda function and configure it to use your deployment package. Then, you publish a function version and create an alias.

1. First, create the Lambda function.
   a. In the AWS Management Console, choose Services, and open the AWS Lambda console.
   b. Choose Create function and then choose Author from scratch.
   c. In the Basic information section, use the following values:
      - For Function name, enter TempMonitor.
      - For Runtime, choose Python 2.7.
      - For Permissions, keep the default setting. This creates an execution role that grants basic Lambda permissions. This role isn't used by AWS IoT Greengrass.
   d. At the bottom of the page, choose Create function.
2. Next, register the handler and upload your Lambda function deployment package.
   a. On the Configuration tab for the TempMonitor function, in Function code, use the following values:
Add a Function to the Group

• For **Code entry type**, choose **Upload a .zip file**.
• For **Runtime**, choose **Python 2.7**.
• For **Handler**, enter `temp_monitor.function_handler`

b. Choose **Upload**.
c. Choose your `temp_monitor_python.zip` deployment package.
d. Choose **Save**.

**Note**
The Test button on the AWS Lambda console doesn't work with this function. The AWS IoT Greengrass Core SDK doesn't contain modules that are required to run your Greengrass Lambda functions independently in the AWS Lambda console. These modules (for example, `greengrass_common`) are supplied to the functions after they are deployed to your Greengrass core.

**Tip**
You can see your code in the **Function code** section by choosing **Edit code inline** from the **Code entry type** menu.

3. Now, publish the first version of your Lambda function and create an alias for the version.

**Note**
Greengrass groups can reference a Lambda function by alias (recommended) or by version. Using an alias makes it easier to manage code updates because you don’t have to change your subscription table or group definition when the function code is updated. Instead, you just point the alias to the new function version.

a. From the **Actions** menu, choose **Publish new version**.
b. For **Version description**, enter **First version** and then choose **Publish**.
c. On the **TempMonitor: 1** configuration page, from the **Actions** menu, choose **Create alias**.
d. On the **Create a new alias** page, use the following values:

• For **Name**, enter `GG_TempMonitor`.
• For **Version**, choose 1.

**Note**
AWS IoT Greengrass doesn't support Lambda aliases for `$LATEST` versions.

e. Choose **Create**.

Now you’re ready to add the Lambda function to your Greengrass group.

**Step 6: Add a Lambda Function to the Greengrass Group**

In this step, you add the Lambda function to the group and then configure its lifecycle and environment variables. For more information, see the section called “Controlling Greengrass Lambda Function Execution” (p. 178).

1. On the group configuration page, choose **Lambdas**, and then choose **Add Lambda**.
Step 7: Add Subscriptions to the Greengrass Group

In this step, you add a subscription that enables the Lambda function to send input data to the connector. The connector defines the MQTT topics that it subscribes to, so this subscription uses one of the topics. This is the same topic that the example function publishes to.

For this tutorial, you also create subscriptions that allow the function to receive simulated temperature readings from AWS IoT and allow AWS IoT to receive status information from the connector.

1. On the group configuration page, choose Subscriptions, and then choose Add Subscription.

2. On the Select your source and target page, configure the source and target, as follows:
   a. For Select a source, choose Lambdas, and then choose TempMonitor.
   b. For Select a target, choose Connectors, and then choose Twilio Notifications.
   c. Choose Next.

3. On the Filter your data with a topic page, for Required topic syntax, choose twilio/txt, and then choose Next.
5. Repeat steps 1 - 4 to create a subscription that allows AWS IoT to publish messages to the function.
   a. For Select a source, choose Services, and then choose IoT Cloud.
   b. For Select a target, choose Lambdas, and then choose TempMonitor.
   c. For Topic filter, enter temperature/input.
6. Repeat steps 1 - 4 to create a subscription that allows the connector to publish messages to AWS IoT.
   a. For Select a source, choose Connectors, and then choose Twilio Notifications.
   b. For Select a target, choose Services, and then choose IoT Cloud.
   c. For Topic filter, twilio/message/status is entered for you. This is the predefined topic that the connector publishes to.

**Step 8: Deploy the Greengrass Group**

Deploy the group to the core device.

1. Make sure that the AWS IoT Greengrass core is running. Run the following commands in your Raspberry Pi terminal, as needed.
   a. To check whether the daemon is running:

   ```bash
   ps aux | grep -E 'greengrass.*daemon'
   ```

   If the output contains a root entry for /greengrass/ggc/packages/ggc-version/bin/daemon, then the daemon is running.

   **Note**
   The version in the path depends on the AWS IoT Greengrass Core software version that's installed on your core device.
   b. To start the daemon:

   ```bash
   cd /greengrass/ggc/core/
   sudo ./greengrassd start
   ```

2. On the group configuration page, choose Deployments, and from the Actions menu, choose Deploy.

3. On the Configure how devices discover your core page, choose Automatic detection.

   This enables devices to automatically acquire connectivity information for the core, such as IP address, DNS, and port number. Automatic detection is recommended, but AWS IoT Greengrass also supports manually specified endpoints. You're only prompted for the discovery method the first time that the group is deployed.
Test the Solution

1. On the AWS IoT console home page, choose Test.

2. For Subscriptions, use the following values, and then choose Subscribe to topic. The Twilio Notifications connector publishes status information to this topic.
3. For **Publish**, use the following values, and then choose **Publish to topic** to invoke the function.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subscription topic</td>
<td>twilio/message/status</td>
</tr>
<tr>
<td>MQTT payload display</td>
<td>Display payloads as strings</td>
</tr>
</tbody>
</table>

### Getting Started with Greengrass Connectors (CLI)

This feature is available for AWS IoT Greengrass Core v1.7 and later.

This tutorial shows how to use the AWS CLI to work with connectors.

Use connectors to accelerate your development life cycle. Connectors are prebuilt, reusable modules that can make it easier to interact with services, protocols, and resources. They can help you deploy business logic to Greengrass devices more quickly. For more information, see [*Integrate with Services and Protocols Using Connectors*](#) (p. 283).
In this tutorial, you configure and deploy the Twilio Notifications connector. The connector receives Twilio message information as input data, and then triggers a Twilio text message. The data flow is shown in the following diagram.

After you configure the connector, you create a Lambda function and a subscription.

- The function evaluates simulated data from a temperature sensor. It conditionally publishes the Twilio message information to an MQTT topic. This is the topic that the connector subscribes to.
- The subscription allows the function to publish to the topic and the connector to receive data from the topic.

The Twilio Notifications connector requires a Twilio auth token to interact with the Twilio API. The token is a text type secret created in AWS Secrets Manager and referenced from a group resource. This enables AWS IoT Greengrass to create a local copy of the secret on the Greengrass core, where it is encrypted and made available to the connector. For more information, see Deploy Secrets to the Core.

The tutorial contains the following high-level steps:

1. Create a Secrets Manager Secret (p. 415)
2. Create a Resource Definition and Version (p. 416)
3. Create a Connector Definition and Version (p. 416)
4. Create a Lambda Function Deployment Package (p. 417)
5. Create a Lambda Function (p. 418)
6. Create a Function Definition and Version (p. 420)
7. Create a Subscription Definition and Version (p. 420)
8. Create a Group Version (p. 421)
9. Create a Deployment (p. 422)

The tutorial should take about 30 minutes to complete.

Using the AWS IoT Greengrass API

It’s helpful to understand the following patterns when you work with Greengrass groups and group components (for example, the connectors, functions, and resources in the group).

- At the top of the hierarchy, a component has a definition object that is a container for version objects. In turn, a version is a container for the connectors, functions, or other component types.
Prerequisites

When you deploy to the Greengrass core, you deploy a specific group version. A group version can contain one version of each type of component. A core is required, but the others are included as needed.

Version are immutable, so you must create new versions when you want to make changes.

Tip
If you receive an error when you run an AWS CLI command, add the --debug parameter and then rerun the command to get more information about the error.

The AWS IoT Greengrass API lets you create multiple definitions for a component type. For example, you can create a FunctionDefinition object every time that you create a FunctionDefinitionVersion, or you can add new versions to an existing definition. This flexibility allows you to customize your version management system.

Prerequisites

To complete this tutorial, you need:

- A Greengrass group and a Greengrass core (v1.7 or later). To learn how to create a Greengrass group and core, see Getting Started with AWS IoT Greengrass (p. 64). The Getting Started tutorial also includes steps for installing the AWS IoT Greengrass Core software.

- AWS IoT Greengrass must be configured to support local secrets, as described in Secrets Requirements (p. 264).

  Note
  This includes allowing access to your Secrets Manager secrets. If you're using the default Greengrass service role, Greengrass has permission to get the values of secrets with names that start with greengrass-.

- A Twilio account SID, auth token, and Twilio-enabled phone number. After you create a Twilio project, these values are available on the project dashboard.

  Note
  You can use a Twilio trial account. If you're using a trial account, you must add non-Twilio recipient phone numbers to a list of verified phone numbers. For more information, see How to Work with your Free Twilio Trial Account.

- AWS CLI installed and configured on your computer. For more information, see Installing the AWS Command Line Interface and Configuring the AWS CLI in the AWS Command Line Interface User Guide.

The examples in this tutorial are written for Linux and other Unix-based systems. If you're using Windows, see Specifying Parameter Values for the AWS Command Line Interface to learn about differences in syntax.

If the command contains a JSON string, the tutorial provides an example that has the JSON on a single line. On some systems, it might be easier to edit and run commands using this format.

Step 1: Create a Secrets Manager Secret

In this step, you use the AWS Secrets Manager API to create a secret for your Twilio auth token.

1. First, create the secret.

   - Replace twilio-auth-token with your Twilio auth token.
Create a Resource Definition and Version

In this step, you use the AWS IoT Greengrass API to create a secret resource for your Secrets Manager secret.

1. Create a resource definition that includes an initial version.
   
   Replace `secret-arn` with the ARN of the secret that you copied in the previous step.

   **JSON Expanded**

   ```
   aws greengrass create-resource-definition --name MyGreengrassResources --initial-version '{
     "Resources": [
       {
         "Id": "TwilioAuthToken",
         "Name": "MyTwilioAuthToken",
         "ResourceDataContainer": {
           "SecretsManagerSecretResourceData": {
             "ARN": "secret-arn"
           }
         }
       }
     ]
   }'
   ```

   **JSON Single-line**

   ```
   aws greengrass create-resource-definition \ 
   --name MyGreengrassResources \ 
   --initial-version '{"Resources": [{"Id": "TwilioAuthToken", "Name": "MyTwilioAuthToken", "ResourceDataContainer": {"SecretsManagerSecretResourceData": {"ARN": "secret-arn"}}}]}'
   ```

2. Copy the `LatestVersionArn` of the resource definition from the output. You use this value to add the resource definition version to the group version that you deploy to the core.

Step 3: Create a Connector Definition and Version

In this step, you configure parameters for the Twilio Notifications connector.
1. Create a connector definition with an initial version.
   - Replace `account-sid` with your Twilio account SID.
   - Replace `secret-arn` with the ARN of your Secrets Manager secret. The connector uses this to get the value of the local secret.
   - Replace `phone-number` with your Twilio-enabled phone number. Twilio uses this to initiate the text message. This can be overridden in the input message payload. Use the following format: +19999999999.

   **JSON Expanded**

   ```json
   aws greengrass create-connector-definition --name MyGreengrassConnectors --initial-version '{
     "Connectors": [
     {
       "Id": "MyTwilioNotificationsConnector",
       "ConnectorArn": "arn:aws:greengrass:region::/connectors/TwilioNotifications/versions/3",
       "Parameters": {
         "TWILIO_ACCOUNT_SID": "account-sid",
         "TwilioAuthTokenSecretArn": "secret-arn",
         "TwilioAuthTokenSecretArn-ResourceId": "TwilioAuthToken",
         "DefaultFromPhoneNumber": "phone-number"
       }
     }
     ]
   }'
   ```

   **JSON Single-line**

   ```bash
   ```

   **Note**
   TwilioAuthToken is the ID that you used in the previous step to create the secret resource.

2. Copy the LatestVersionArn of the connector definition from the output. You use this value to add the connector definition version to the group version that you deploy to the core.

**Step 4: Create a Lambda Function Deployment Package**

To create a Lambda function, you must first create a Lambda function deployment package that contains the function code and dependencies. Greengrass Lambda functions require the AWS IoT Greengrass Core SDK (p. 175) for tasks such as communicating with MQTT messages in the core environment and accessing local secrets. This tutorial creates a Python function, so you use the Python version of the SDK in the deployment package.
1. Download the AWS IoT Greengrass Core SDK for Python from the AWS IoT Greengrass Core SDK (p. 22) downloads page.
2. Unzip the downloaded package to get the SDK. The SDK is the greengrassdk folder.
3. Save the following Python code function in a local file named temp_monitor.py.

```python
from __future__ import print_function
import greengrasssdk
import json
import random

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

def function_handler(event, context):
    temp = event['temperature']

    # check the temperature
    # if greater than 30C, send a notification
    if temp > 30:
        data = build_request(event)
        client.publish(topic='twilio/txt', payload=json.dumps(data))
        print('published:' + str(data))
        print('temperature:' + str(temp))
        return

    # build the Twilio request from the input data
    def build_request(event):
        to_name = event['to_name']
        to_number = event['to_number']
        temp_report = 'temperature:' + str(event['temperature'])

        return {
            "request": {
                "recipient": {
                    "name": to_name,
                    "phone_number": to_number,
                    "message": temp_report
                }
            },
            "id": "request_" + str(random.randint(1,101))
        }
```

4. Zip the following items into a file named temp_monitor_python.zip. When creating the ZIP file, include only the code and dependencies, not the containing folder.
   - temp_monitor.py. App logic.
   - greengrasssdk. Required library for Python Greengrass Lambda functions that publish MQTT messages.

This is your Lambda function deployment package.

**Step 5: Create a Lambda Function**

Now, create a Lambda function that uses the deployment package.

1. First, create an IAM role so you can pass in the ARN when you create the function.
Create a Lambda Function

**Note**

AWS IoT Greengrass doesn't use this role because permissions for your Greengrass Lambda functions are specified in the Greengrass group role. For this tutorial, you create an empty role, or alternatively, you can use an existing execution role.

### JSON Expanded

```bash
aws iam create-role --role-name Lambda_empty --assume-role-policy '{
  "Version": "2012-10-17",
  "Statement": [
    { "Effect": "Allow",
      "Principal": { "Service": "lambda.amazonaws.com" },
      "Action": "sts:AssumeRole"
    }
  ]
}"
```

### JSON Single-line

```bash
aws iam create-role --role-name Lambda_empty --assume-role-policy '{"Version": "2012-10-17", "Statement": [{"Effect": "Allow", "Principal": {"Service": "lambda.amazonaws.com"},"Action": "sts:AssumeRole"}]}'
```

2. Copy the Arn of the output.
3. Use the AWS Lambda API to create the TempMonitor function. The following command assumes that the zip file is in the current directory.

   - Replace `role-arn` with the Arn that you copied.

   ```bash
   aws lambda create-function \
   --function-name TempMonitor \
   --zip-file fileb://temp_monitor_python.zip \
   --role role-arn \
   --handler temp_monitor.function_handler \
   --runtime python2.7
   ```

4. Publish a version of the function.

   ```bash
   aws lambda publish-version --function-name TempMonitor --description 'First version'
   ```

5. Create an alias for the published version.

   Greengrass groups can reference a Lambda function by alias (recommended) or by version. Using an alias makes it easier to manage code updates because you don't have to change your subscription table or group definition when the function code is updated. Instead, you just point the alias to the new function version.

   **Note**

   AWS IoT Greengrass doesn't support Lambda aliases for $LATEST versions.

   ```bash
   aws lambda create-alias --function-name TempMonitor --name GG_TempMonitor --function-version 1
   ```
Create a Function Definition and Version

To use a Lambda function on an AWS IoT Greengrass core, you create a function definition version that references the Lambda function by alias and defines the group-level configuration. For more information, see the section called “Controlling Greengrass Lambda Function Execution” (p. 178).

1. Create a function definition that includes an initial version.

   - Replace `alias-arn` with the `AliasArn` that you copied when you created the alias.

   JSON Expanded

   ```bash
   aws greengrass create-function-definition --name MyGreengrassFunctions --initial-version '{
     "Functions": [
       {
         "Id": "TempMonitorFunction",
         "FunctionArn": "alias-arn",
         "FunctionConfiguration": {
           "Executable": "temp_monitor.function_handler",
           "MemorySize": 16000,
           "Timeout": 5
         }
       }
     ]
   }'
   ```

   JSON Single-line

   ```bash
   aws greengrass create-function-definition --name MyGreengrassFunctions --initial-version '"
"Functions": {"Id": "TempMonitorFunction", "FunctionArn": "alias-arn", "FunctionConfiguration": {"
"Executable": "temp_monitor.function_handler", "MemorySize": 16000, "Timeout": 5}}}'
   ```

2. Copy the `LatestVersionArn` from the output. You use this value to add the function definition version to the group version that you deploy to the core.

3. Copy the `Id` from the output. You use this value later when you update the function.

Step 7: Create a Subscription Definition and Version

In this step, you add a subscription that enables the Lambda function to send input data to the connector. The connector defines the MQTT topics that it subscribes to, so this subscription uses one of the topics. This is the same topic that the example function publishes to.

For this tutorial, you also create subscriptions that allow the function to receive simulated temperature readings from AWS IoT and allow AWS IoT to receive status information from the connector.

1. Create a subscription definition that contains an initial version that includes the subscriptions.
• Replace `alias-arn` with the `AliasArn` that you copied when you created the alias for the function. Use this ARN for both subscriptions that use it.

### JSON Expanded

```bash
aws greengrass create-subscription-definition --initial-version '{
    "Subscriptions": [
        {
            "Id": "TriggerNotification",
            "Source": "alias-arn",
            "Subject": "twilio/txt",
            "Target": "arn:aws:greengrass:region::/connectors/TwilioNotifications/versions/3"
        },
        {
            "Id": "TemperatureInput",
            "Source": "cloud",
            "Subject": "temperature/input",
            "Target": "alias-arn"
        },
        {
            "Id": "OutputStatus",
            "Source": "arn:aws:greengrass:region::/connectors/TwilioNotifications/versions/3",
            "Subject": "twilio/message/status",
            "Target": "cloud"
        }
    ]
}
```

### JSON Single-line

```bash
```

2. Copy the `LatestVersionArn` from the output. You use this value to add the subscription definition version to the group version that you deploy to the core.

### Step 8: Create a Group Version

Now, you're ready to create a group version that contains all of the items that you want to deploy. You do this by creating a group version that references the target version of each component type.

First, get the group ID and the ARN of the core definition version. These values are required to create the group version.

1. Get the ID of the group:
   a. List your groups.
Step 9: Create a Deployment

Deploy the group to the core device.

1. In a core device terminal, make sure that the AWS IoT Greengrass daemon is running.
   a. To check whether the daemon is running:
Test the Solution

1. On the AWS IoT console home page, choose Test.
2. For **Subscriptions**, use the following values, and then choose **Subscribe to topic**. The Twilio Notifications connector publishes status information to this topic.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subscription topic</td>
<td>twilio/message/status</td>
</tr>
<tr>
<td>MQTT payload display</td>
<td>Display payloads as strings</td>
</tr>
</tbody>
</table>

3. For **Publish**, use the following values, and then choose **Publish to topic** to invoke the function.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topic</td>
<td>temperature/input</td>
</tr>
</tbody>
</table>
| Message          | Replace `recipient-name` with a name and `recipient-phone-number` with the phone number of the text message recipient. Example: +12345000000<br>```json
{
  "to_name": "recipient-name",
  "to_number": "recipient-phone-number",
  "temperature": 31
}
```

If you're using a trial account, you must add non-Twilio recipient phone numbers to a list of...
<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>verified phone numbers. For more information, see <a href="#">Verify your Personal Phone Number</a>.</td>
</tr>
</tbody>
</table>

If successful, the recipient receives the text message and the console displays the **success** status from the output data (p. 401).

Now, change the **temperature** in the input message to **29** and publish. Because this is less than 30, the TempMonitor function doesn’t trigger a Twilio message.

**See Also**

- [Integrate with Services and Protocols Using Connectors](#) (p. 283)
- the section called “AWS-Provided Greengrass Connectors” (p. 288)
- AWS Secrets Manager commands in the [AWS CLI Command Reference](#)
- IAM commands in the [AWS CLI Command Reference](#)
- AWS Lambda commands in the [AWS CLI Command Reference](#)
- AWS IoT Greengrass commands in the [AWS CLI Command Reference](#)
Greengrass Discovery RESTful API

All devices that communicate with an AWS IoT Greengrass core must be a member of a Greengrass group. Each group must have an AWS IoT Greengrass core. The Discovery API enables devices to retrieve information required to connect to an AWS IoT Greengrass core that is in the same Greengrass group as the device. When a device first comes online, it can connect to the AWS IoT Greengrass cloud service and use the Discovery API to find:

- The group to which it belongs. A device can be a member of up to 10 groups.
- The IP address and port for the AWS IoT Greengrass core in the group.
- The group's root CA certificate, which can be used to authenticate the AWS IoT Greengrass core device.

To use this API, send HTTP requests to the Discovery API endpoint. For example:

```
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 Regions and Endpoints in the AWS General Reference. This is a data plane only API. The endpoints for group management and AWS IoT operations are different from the Discovery API endpoints.

### 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 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” (p. 46).

**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
```

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.

**Note**

These examples use the Amazon Trust Services (ATS) endpoint, which is used with ATS root CA certificates (recommended). Endpoints must match the root CA certificate type.
For more information, see the section called “Endpoints Must Match the Certificate Type” (p. 45).

Response

Upon success, the response includes the standard HTTP headers plus the following code and body:

```
HTTP 200
BODY: response document
```

For more information, see Example Discover Response Documents (p. 427).

Authorization

Retrieving the connectivity information requires a policy that allows the caller to perform the \texttt{greengrass:Discover} action. TLS mutual authentication with a client certificate is the only accepted form of authentication. The following is an example policy that allows a caller to perform this action:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": "greengrass:Discover",
        "Resource": 
            ["arn:aws:iot:us-west-2:123456789012:thing/MyThingName"]
    }]
}
```

Example Discover Response Documents

The following document shows the response for a device that is a member of a group with one AWS IoT Greengrass core, one endpoint, and one group CA:

```
{
    "GGGroups": [
        { 
            "GGGroupId": "gg-group-01-id",
            "Cores": [
                {
                    "thingArn": "core-01-thing-arn",
                    "Connectivity": [
                        {
                            "id": "core-01-connection-id",
                            "hostAddress": "core-01-address",
                            "portNumber": "core-01-port",
                            "metadata": "core-01-description"
                        }
                    ]
                }
            ],
            "CAs": ["-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----"]
        }
    ]
}
```
The following document shows the response for a device that is a member of two groups with one AWS IoT Greengrass core, multiple endpoints, and multiple group CAs:

```json
{
    "GGGroups": [
        {
            "GGGroupId": "gg-group-01-id",
            "Cores": [
                {
                    "thingArn": "core-01-thing-arn",
                    "Connectivity": [
                        {
                            "id": "core-01-connection-id",
                            "hostAddress": "core-01-address",
                            "portNumber": core-01-port,
                            "metadata": "core-01-connection-1-description"
                        },
                        {
                            "id": "core-01-connection-id-2",
                            "hostAddress": "core-01-address-2",
                            "portNumber": core-01-port-2,
                            "metadata": "core-01-connection-2-description"
                        }
                    ]
                }
            ],
            "CAs": [
                "-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----",
                "-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----",
                "-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----"
            ]
        },
        {
            "GGGroupId": "gg-group-02-id",
            "Cores": [
                {
                    "thingArn": "core-02-thing-arn",
                    "Connectivity": [
                        {
                            "id": "core-02-connection-id",
                            "hostAddress": "core-02-address",
                            "portNumber": core-02-port,
                            "metadata": "core-02-connection-1-description"
                        }
                    ],
                    "CAs": [
                        "-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----",
                        "-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----",
                        "-----BEGIN CERTIFICATE-----cert-contents-----END CERTIFICATE-----"
                    ]
                }
            ]
        }
    ]
}
```

**Note**

An AWS IoT Greengrass group must define exactly one AWS IoT Greengrass core. Any response from the AWS IoT Greengrass cloud service that contains a list of AWS IoT Greengrass cores contains only one AWS IoT Greengrass core.

If you have `cURL` installed, you can test the discovery request. For example:
$ curl --cert 1a23bc4d56.cert.pem --key 1a23bc4d56.private.key https://greengrass-ats.iot.us-west-2.amazonaws.com:8443/greengrass/discover/thing/MyDevice
{"GGGroups":[{"GGGroupId":"1234a5b6-78cd-901e-2fgh-3i45j6k1789","Cores":[{"thingArn":"arn:aws:iot:us-west-2:12345689012:thing/MyFirstGroup_Core","Connectivity":[{"Id":"AUTOIP_192.168.1.4_1","HostAddress":"192.168.1.5","PortNumber":8883,"Metadata":""}]}]},"CAs":[]}
Implement Custom OPC-UA Support to Communicate with Industrial Equipment

OPC-UA is an information exchange standard for industrial communication. You can create custom implementations for AWS IoT Greengrass that allow you to use OPC-UA to ingest and process messages from industrial equipment. Based on rules you define, messages can be delivered to local or cloud targets.

This tutorial shows an example custom OPC-UA implementation on your AWS IoT Greengrass core. The example is based on an open source implementation that supports certificate-based authentication and is fully customizable. You can follow this pattern to add support for OPC-UA and other custom, legacy, or proprietary messaging protocols.

Note
This example is a lightweight implementation that handles limited amounts of data. We recommend that you use AWS IoT SiteWise (currently in limited preview) to collect and organize industrial data at scale. For more information, see What Is AWS IoT SiteWise? in the AWS IoT SiteWise User Guide.

The example implementation uses a Node.js Lambda function that acts as an OPC-UA proxy on a Greengrass core. This is possible because Lambda functions running on a core can access network resources. In the following diagram, the OPC-UA_Adapter Lambda function transfers information sent from an OPC-UA server over TCP to other functions or services in the Greengrass group.

In this tutorial, you create a Lambda function that connects to an OPC-UA server, monitors an OPC-UA node in the server, and gets a message when the value of a monitored node changes. You configure a long-lived connection between the Lambda function and your OPC-UA server and use OPC-UA subscriptions that allow the function to monitor changes to predefined nodes. Changes to these nodes trigger a Publish event from the OPC-UA server. When the Lambda function receives messages from the server, it republishes the messages to predefined topics.

The topic structure uses the following syntax:
The tutorial contains the following high-level steps:

1. the section called “Set Up a Test OPC-UA Server” (p. 431)
2. the section called “Create an OPC-UA Adapter Lambda Function That Interacts with the OPC-UA Server” (p. 432)
3. the section called “Test the OPC-UA Adapter Lambda Function” (p. 434)

Prerequisites

- A Greengrass group and a Greengrass core. To learn how to create a Greengrass group or core, see *Getting Started with AWS IoT Greengrass* (p. 64).
- Node.js 8.10 installed on your core device.

Set Up a Test OPC-UA Server

- Use the following commands to set up a test OPC-UA server. Skip this step if you have an OPC-UA server you want to use.

  **Note**  
  The npm package manager must be installed to run this command. For information, see Get npm on the npm website.

  ```
git clone git://github.com/node-opcua/node-opcua.git  
cd node-opcua  
git checkout v0.0.65  
npm install  
node bin/simple_server
```

The server produces the following output:

```
[ec2-user@your_instance_id node-opcua]$ node bin/simple_server
  server PID : 28585
registering server to :opc.tcp://your_instance_id4840/UADiscovery
err Cannot find module 'usage'
skipping installation of cpu_usage and memory_usage nodes
  server on port : 26543
  endpointUrl : opc.tcp://your_instance_idregion.compute.internal:26543
  serverInfo :
    applicationUri : urn:54f7890cca4c49a1:NodeOPCUA-Server
    productUri : NodeOPCUA-Server
    applicationName : locale=en text=NodeOPCUA
    applicationType : SERVER
```
Create an OPC-UA Adapter Lambda Function That Interacts with the OPC-UA Server

1. Prepare your Lambda function.

Get the code for an OPC-UA adapter Lambda function from GitHub:

```bash
git clone https://github.com/aws-samples/aws-greengrass-samples.git
cd aws-greengrass-samples/greengrass-opcua-adapter-nodejs
npm install
```

**Note**
The `node-opcua` library (v0.0.65) used by the Lambda function attempts to regenerate some model files at runtime. Because Lambda functions start with a containerized, read-only file system on a Greengrass core, the attempt to generate other code fails. The next step fixes this issue.

2. Change line 109 of the file at `node_modules/node-opcua/lib/misc/factories.js` to this:

```javascript
var generated_source_is_outdated = (!generated_source_exists);
```

Run this command to make that change:

```bash
sed -i '109s/.*/    var generated_source_is_outdated = (!generated_source_exists);/ node_modules/node-opcua/lib/misc/factories.js
```

3. Configure the server and monitored nodes.

Change the `configSet` variable inside the `index.js` file of the OPC-UA adapter Lambda function to use the target server IP address and port, and the node IDs you want to monitor. By default, it uses the following configuration:

```javascript
const configSet = {
    server: {
        name: 'server',
        url: 'opc.tcp://localhost:26543',
    },
    subscriptions: [
        
    ]
```
null

In this case, you connect to an OPC-UA server running on the same host as your Greengrass core (on port 26543) and monitor the node with the OPC-UA ID ‘ns=1;s=PumpSpeed’.

4. Configure the authentication mode.

The OPC-UA library used in this example supports three modes of authentication to your OPC-UA server. The most secure method is certificate-based authentication, but the library also supports user name and password or no authentication.

To set certificate-based authentication:

- Package your certificate and private key with your Lambda function (for example, under a directory named certs/).
- Change the `clientOptions` variable to contain `certificateFile`, `privateKeyFile`, `securityModes`, and `securityPolicies` options:

```javascript
const clientOptions = {
  keepSessionAlive: true,
  certificateFile: '/lambda/certs/certificate_name.pem.crt',
  privateKeyFile: '/lambda/certs/private_key_name.pem.key',
  securityModes: MessageSecurityMode.SIGN,
  securityPolicies: SecurityPolicy.BASIC256,
  connectionStrategy: {
    maxRetry: 1000000,
    initialDelay: 2000,
    maxDelay: 10 * 1000,
  },
};
```

5. Download the AWS IoT Greengrass Core SDK for Node.js from the AWS IoT Greengrass Core SDK (p. 22) downloads page.

6. Create and upload your Lambda function.

Run the following commands to create the Lambda function and upload the code and dependencies. For more information, see Configure the Lambda Function for AWS IoT Greengrass (p. 88).

```bash
# Install Greengrass SDK in the node_modules directory. If you used the curl command to download the SDK, replace the tar.gz file name # with the name you used.
tar -zxvf aws-greengrass-core-sdk-js-*.tar.gz -C /tmp/
unzip /tmp/aws_greengrass_core_sdk_js/sdk/aws-greengrass-core-sdk-js.zip -d
node_modules

# Archive the whole directory as a zip file
zip -r opcuaLambda.zip * -x */.git/*

# Create an AWS Lambda function with the zip file
aws lambda create-function --function-name function-name --runtime 'nodejs8.10' --role your_role --handler 'index.handler' --zip-file opcuaLambda.zip
```

7. Configure and deploy the Lambda function.

Add the Lambda function to your Greengrass group and configure its group-level settings. For instructions, see Configure the Lambda Function for AWS IoT Greengrass (p. 88).
Test the OPC-UA Adapter Lambda Function

The Lambda function should start receiving messages from your OPC-UA server after you deploy the group.

- If you’re using your own OPC-UA server, trigger a change in the OPC-UA node ID that you specified and check if your Lambda function receives a message.
- If you’re using the example server, the PumpSpeed node is configured to simulate a series of consecutive updates, so your Lambda function should receive multiple messages per second.

You can see the messages received by your Lambda function in one of two ways:

- Check the Lambda function's logs:

  ```
  sudo cat /greengrass-root/ggc/var/log/user/region/account-id/your_function_name.log
  ```

  The logs should look similar to this example:

  ```
  [2017-11-14T16:33:09.05Z][INFO]-started subscription : 305964
  [2017-11-14T16:33:09.05Z][INFO]-monitoring node id = ns=1;s=PumpSpeed
  [2017-11-14T16:33:09.099Z][INFO]-monitoredItem initialized
  [2017-11-15T23:49:34.752Z][INFO]-Publishing message on topic "/opcua/server/node/MyPumpSpeed" with Payload "{"id":"ns=1;s=PumpSpeed","value":{"dataType":"Double","arrayType":"Scalar","value":237.5250759433095}}"
  ```

- In the AWS IoT console, create a subscription that allows your Lambda function to publish messages to AWS IoT:

  Set your Lambda function as the source and **IoT Cloud** as the target. For instructions, see [this step](p. 91).

  Follow the steps in the section called “Verify the Lambda Function Is Running on the Device” (p. 94) to view the messages in the AWS IoT console.

**Note**
To avoid incurring charges for message sent to AWS IoT after you finish testing, stop the Greengrass daemon, and then terminate the example server.
To stop the AWS IoT Greengrass daemon:

```
    cd /greengrass-root/ggc/core/
    sudo ./greengrassd stop
```
Next Steps

Use this pattern to create your own implementation to support OPC-UA and other custom, legacy, or proprietary messaging protocols. Your implementation can use local Lambda functions to access network resources and add support for any protocol on top of TCP/IP. You can also use Greengrass local resources to add support for protocols that need access to hardware adapters and drivers. Or, to learn about AWS IoT SiteWise (currently in limited preview), an industry-level solution designed to ingest data at scale, see What Is AWS IoT SiteWise? in the AWS IoT SiteWise User Guide.
AWS IoT Greengrass Security

AWS IoT Greengrass uses X.509 certificates, managed subscriptions, AWS IoT policies, and IAM policies and roles to secure the applications that run on devices in your local Greengrass environment.

Devices in Greengrass environments fall into one of the following two categories. Both device types require an entry in the AWS IoT registry, a device certificate, and an AWS IoT policy.

- **AWS IoT Greengrass cores.** Core devices use certificates and policies to securely connect to AWS IoT. The certificates and policies also allow AWS IoT Greengrass to deploy configuration information, Lambda functions, connectors, and managed subscriptions to core devices.

- **AWS IoT devices that connect to a Greengrass core.** These Greengrass devices use certificates and policies to securely connect to AWS IoT and AWS IoT Greengrass services. This allows devices to use the Greengrass Discovery service to find and connect to a core device. A Greengrass device uses the same certificate to connect to the AWS IoT device gateway and core device.

The following diagram shows the components of the AWS IoT Greengrass security model:

A - Greengrass service role

A customer-created IAM role that allows AWS IoT Greengrass access to your AWS IoT, Lambda, and other AWS resources. For more information, see the section called “Greengrass Service Role” (p. 443).

B - Core device certificate

An X.509 certificate used to authenticate an AWS IoT Greengrass core.

C - Device certificate

An X.509 certificate used to authenticate an AWS IoT Greengrass device.
Configuring Greengrass Security

To configure your Greengrass application's security:

1. Create an AWS IoT thing for your AWS IoT Greengrass core device.
2. Generate a key pair and device certificate for your AWS IoT Greengrass core device.
3. Create and attach an AWS IoT policy to the device certificate. The certificate and policy allow the AWS IoT Greengrass core device access to AWS IoT and AWS IoT Greengrass services. For more information, see the section called “Minimal AWS IoT Policy for the AWS IoT Greengrass Core Device” (p. 437).
4. Create a Greengrass service role. This IAM role authorizes AWS IoT Greengrass to access resources from other AWS services on your behalf. This allows AWS IoT Greengrass to perform essential tasks, such as retrieving AWS Lambda functions and managing AWS IoT shadows. You can use the same service role across AWS Regions, but it must be associated with every AWS Region where you use AWS IoT Greengrass. For more information, see the section called “Greengrass Service Role” (p. 443).
5. (Optional) Create a Greengrass group role. This IAM role grants permission to Lambda functions and connectors running on an AWS IoT Greengrass core to call AWS services. For example, the Kinesis Firehose connector (p. 307) requires permission to write records to an Amazon Kinesis Data Firehose delivery stream. You create a separate group role for each Greengrass group.
6. Create an AWS IoT thing for each device that will connect to your AWS IoT Greengrass core.
7. Create device certificates, key pairs, and AWS IoT policies for each device that connects to your AWS IoT Greengrass core.

Note
You can also use existing AWS IoT things and certificates.

Minimal AWS IoT Policy for the AWS IoT Greengrass Core Device

An AWS IoT policy defines the set of actions allowed for an AWS IoT thing (in this case, the Greengrass core device). The policy is attached to the device certificate and used to access AWS IoT and AWS IoT Greengrass services. AWS IoT policies are JSON documents that follow the conventions of IAM policies. For more information, see AWS IoT Policies in the AWS IoT Developer Guide.

The following example policy includes the minimum set of actions required to support basic Greengrass functionality for your core device. Make a note of the following:
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, Lambda functions, connectors, and devices in the Greengrass group, specify the topics and topic filters that you want to allow. For more information, see Publish/Subscribe Policy Examples in the AWS IoT Developer Guide.

The policy includes a section that allows AWS IoT to get, update, and delete the core device's shadow. To allow shadow sync for other devices in the Greengrass group, specify the target ARNs in the Resource list (for example, arn:aws:iot:region:account-id:thing/device-name).

For the greengrass:UpdateCoreDeploymentStatus permission, the final segment in the Resource ARN is the URL-encoded ARN of the core device.

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:Connect"
      ],
      "Resource": [
        "*
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iot:Publish",
        "iot:Subscribe",
        "iot:Receive"
      ],
      "Resource": [
        "arn:aws:iot:region:account-id:topicfilter/$aws/things/core-name-*",
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "iot:GetThingShadow",
        "iot:UpdateThingShadow",
        "iot:DeleteThingShadow"
      ],
      "Resource": [
        "arn:aws:iot:region:account-id:thing/core-name-*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "greengrass:AssumeRoleForGroup",
        "greengrass:CreateCertificate"
      ],
      "Resource": [
        "*
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "greengrass:GetDeployment"
      ],
      "Resource": [  
```
On the AWS IoT console, you can easily view and edit the policy that's attached to your core's certificate.

1. In the navigation pane, choose Manage, choose Things, and then choose your core.
2. On your core's configuration page, choose Security.
3. On the Certificates page, choose your certificate.
4. On the certificate's configuration page, choose Policies, and then choose the policy.

If you want to edit the policy, choose Edit policy document.

AWS IoT Greengrass Core Security Principals

The AWS IoT Greengrass core uses the following security principals: AWS IoT client, local MQTT server, and local secrets manager. The configuration for these principals is stored in the crypto object in the config.json configuration file. For more information, see the section called “AWS IoT Greengrass Core Configuration File” (p. 24).

This configuration includes the path to the private key used by the principal component for authentication and encryption. AWS IoT Greengrass supports two modes of private key storage: hardware-based or file system-based (default). For more information about storing keys on hardware security modules, see the section called “Hardware Security” (p. 448).

AWS IoT Client

The AWS IoT client manages communication over the internet between the Greengrass core and AWS IoT. AWS IoT Greengrass uses X.509 certificates with public and private keys for mutual
authentication when establishing TLS connections for this communication. For more information, see X.509 Certificates and AWS IoT in the AWS IoT Developer Guide.

The IoT client supports RSA and EC certificates and keys. The certificate and private key path are specified for the IoTCertificate principal in config.json.

**MQTT Server**

The local MQTT server manages communication over the local network between the Greengrass core and other Greengrass devices in the group. AWS IoT Greengrass uses X.509 certificates with public and private keys for mutual authentication when establishing TLS connections for this communication.

By default, AWS IoT Greengrass generates an RSA private key for you. To configure the core to use a different private key, you must provide the key path for the MQTTServerCertificate principal in config.json.

**Private Key Support**

<table>
<thead>
<tr>
<th></th>
<th>RSA Key</th>
<th>EC Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key type</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>Key parameters</td>
<td>Minimum 2048-bit length</td>
<td>NIST P-256 or NIST P-384 curve</td>
</tr>
<tr>
<td>Disk format</td>
<td>PKCS#1, PKCS#8</td>
<td>SECG1, PKCS#8</td>
</tr>
</tbody>
</table>
| Minimum GGC version  | • Use default RSA key: 1.0  
                    | • Specify an RSA key: 1.7  | • Specify an EC key: 1.9  |

The configuration of the private key determines related processes. For the list of cipher suites that the Greengrass core supports as a server, see the section called "TLS Cipher Suites Support" (p. 442).

**If no private key is specified** (default)

- AWS IoT Greengrass rotates the key based on your rotation settings.
- The core generates an RSA key, which is used to generate the certificate.
- The MQTT server certificate has an RSA public key and an SHA-256 RSA signature.

**If an RSA private key is specified** (requires GGC v1.7 or later)

- You are responsible for rotating the key.
- The core uses the specified key to generate the certificate.
- The RSA key must have a minimum length of 2048 bits.
- The MQTT server certificate has an RSA public key and an SHA-256 RSA signature.

**If an EC private key is specified** (requires GGC v1.9)

- You are responsible for rotating the key.
- The core uses the specified key to generate the certificate.
- The EC private key must use an NIST P-256 or NIST P-384 curve.
- The MQTT server certificate has an EC public key and an SHA-256 RSA signature.

The MQTT server certificate presented by the core has an SHA-256 RSA signature, regardless of the key type. For this reason, clients must support SHA-256 RSA certificate validation to establish a secure connection with the core.
Secrets Manager

The local secrets manager securely manages local copies of secrets that you create in AWS Secrets Manager. It uses a private key to secure the data key that’s used to encrypt the secrets. For more information, see Deploy Secrets to the Core (p. 263).

By default, the IoT client private key is used, but you can specify a different private key for the SecretsManager principal in config.json. Only the RSA key type is supported. For more information, see the section called “Specify the Private Key for Secret Encryption” (p. 265).

Note
Currently, AWS IoT Greengrass supports only the PKCS#1 v1.5 padding mechanism for encryption and decryption of local secrets when using hardware-based private keys. If you’re following vendor-provided instructions to manually generate hardware-based private keys, make sure to choose PKCS#1 v1.5. AWS IoT Greengrass doesn’t support Optimal Asymmetric Encryption Padding (OAEP).

Private Key Support

<table>
<thead>
<tr>
<th></th>
<th>RSA Key</th>
<th>EC Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key type</td>
<td>Supported</td>
<td>Not supported</td>
</tr>
<tr>
<td>Key parameters</td>
<td>Minimum 2048-bit length</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Disk format</td>
<td>PKCS#1, PKCS#8</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Minimum GGC version</td>
<td>1.7</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

Device Connection Workflow

This section describes how devices connect to the AWS IoT Greengrass cloud service and AWS IoT Greengrass core devices.

• An AWS IoT Greengrass core device uses its device certificate, private key, and the AWS IoT root CA certificate to connect to the Greengrass cloud service.
• The AWS IoT Greengrass core device downloads group membership information from the Greengrass service.
• When a deployment is made to the AWS IoT Greengrass core device, the Device Certificate Manager (DCM) handles certificate management for the AWS IoT Greengrass core device.
• An AWS IoT device connects to the Greengrass cloud service using its device certificate, private key, and the AWS IoT root CA. After making the connection, the AWS IoT device uses the Greengrass Discovery Service to find the IP address of its AWS IoT Greengrass core device. The device can also download the group’s root CA certificate, which can be used to authenticate the Greengrass core device.
• An AWS IoT device attempts to connect to the AWS IoT Greengrass core, passing its device certificate and client ID. If the client ID matches the thing name of the device and the certificate is valid, the connection is made. Otherwise, the connection is terminated.

Greengrass Messaging Workflow

AWS IoT Greengrass uses a subscription table to define how MQTT messages can be exchanged between devices, functions, and connectors in a Greengrass group, and with AWS IoT or the local shadow service. Each subscription specifies a source, target, and MQTT topic (or subject) over which messages are
sent or received. AWS IoT Greengrass allows messages to be sent from a source to a target only if a corresponding subscription is defined.

A subscription defines the message flow in one direction only, from the source to the target. To support two-way message exchange, you must create two subscriptions, one for each direction.

Certificate Rotation on the MQTT Core Server

Greengrass devices use the MQTT core server certificate to authenticate with the Greengrass core device. By default, this certificate expires in seven days. Its lifetime is limited for security reasons, so that if unauthorized parties obtain the certificate, it cannot be used for long.

For certificate rotation to occur, your Greengrass-enabled device must be online and able to access the Greengrass cloud service directly on a regular basis. When the certificate expires, the Greengrass core device attempts to connect to the Greengrass cloud service to obtain a new certificate. If the connection is successful, the core device downloads a new MQTT core server certificate and restarts the local MQTT service. At this point, all Greengrass devices connected to the core are disconnected. If the device is offline at the time of expiry, it does not receive the replacement certificate. Any new attempts to connect to the core device are rejected. Existing connections are not affected. Devices cannot connect to the core until the connection to the Greengrass cloud service is restored and a new MQTT core server certificate can be downloaded.

You can set the expiration to any value between 7 and 30 days, depending on your needs. More frequent rotation requires more frequent cloud connection. Less frequent rotation can pose security concerns. If you want to set the certificate expiration to a value higher than 30 days, contact AWS Support.

When the MQTT core server certificate expires, any attempt to validate the certificate fails. The device must be able to detect the failure and terminate the connection.

TLS Cipher Suites Support

AWS IoT Greengrass uses the AWS IoT transport security model to encrypt communication with the cloud by using TLS cipher suites. In addition, AWS IoT Greengrass data is encrypted when at rest (in the cloud). For more information about AWS IoT transport security and supported cipher suites, see Transport Security in the AWS IoT Developer Guide.

Supported Cipher Suites for Local Network Communication

As opposed to the AWS IoT cloud, the AWS IoT Greengrass core supports the following local network TLS cipher suites for certificate-signing algorithms. All of these cipher suites are supported when private keys are stored on the file system. A subset are supported when the core is configured to use hardware security modules (HSM). For more information, see the section called “Security Principals” (p. 439) and the section called “Hardware Security” (p. 448). The table also includes the minimum version of AWS IoT Greengrass Core software required for support.

<table>
<thead>
<tr>
<th>Cipher</th>
<th>HSM Support</th>
<th>Minimum GGC Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLSv1.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA</td>
<td>Supported</td>
<td>1.0</td>
</tr>
<tr>
<td>TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA</td>
<td>Supported</td>
<td>1.0</td>
</tr>
<tr>
<td>TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384</td>
<td>Supported</td>
<td>1.0</td>
</tr>
<tr>
<td>TLS_RSA_WITH_AES_128_CBC_SHA</td>
<td>Not supported</td>
<td>1.0</td>
</tr>
<tr>
<td>TLS_RSA_WITH_AES_128_GCM_SHA256</td>
<td>Not supported</td>
<td>1.0</td>
</tr>
</tbody>
</table>
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 makes it possible for AWS IoT Greengrass to perform essential tasks, such as retrieving your AWS Lambda functions and managing AWS IoT shadows.

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 define equivalent permissions. This policy is maintained by AWS and defines the set of permissions required by AWS IoT Greengrass.

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. Group deployment fails if the service role doesn’t exist in the current AWS account and Region.

The following sections describe how to create and manage the Greengrass service role in the AWS Management Console or AWS CLI.

- Manage the Service Role (Console) (p. 443)
- Manage the Service Role (CLI) (p. 446)

**Note**
In addition to the service role that authorizes service-level access, you can assign a group role to an AWS IoT Greengrass group. The group role is a separate IAM role that controls how Greengrass Lambda functions and connectors in the group can access AWS services.

### Managing the Greengrass Service Role (Console)

The AWS IoT console makes it easy to manage your Greengrass service role. For example, when you create or deploy a Greengrass group, the console checks whether your AWS account is attached to a
Greengrass service role in the AWS Region that's currently selected in the console. 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” (p. 444).

You can use the AWS IoT console for the following role management tasks:

- Find Your Greengrass Service Role (p. 444)
- Create the Greengrass Service Role (p. 444)
- Change the Greengrass Service Role (p. 445)
- Detach the Greengrass Service Role (p. 445)

**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 is using in the current AWS Region.

1. In the AWS IoT console, in the navigation pane, choose **Settings**.
2. Scroll to the Greengrass service role section to see your service role and its policies.

If you don't see a service role, you can let the console create or configure one for you. For more information, see Create the Greengrass Service Role (p. 444).

---

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.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Greengrass_ServiceRole</td>
</tr>
<tr>
<td>Property</td>
<td>Value</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>Trusted entity</td>
<td>AWS service: greengrass</td>
</tr>
<tr>
<td>Policy</td>
<td>AWSGreengrassResourceAccessRolePolicy</td>
</tr>
</tbody>
</table>

When you create or deploy a Greengrass group from the AWS IoT console, the console checks whether a Greengrass service role is associated with your AWS account in the AWS Region that's currently selected in the console. 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 different service role or use custom role policies, you can 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.

**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. In the AWS IoT console, in the navigation pane, choose Settings.
2. Under Greengrass service role, choose Choose different role.
   
   The IAM roles in your AWS account that define AWS IoT Greengrass as a trusted entity are displayed in the Choose the Greengrass service role dialog box.
3. Choose your Greengrass service role.
4. Choose Save.

**Note**
To allow the console to create a default Greengrass service role for you, choose Create role for me instead of choosing a role from the list. The Create role for me link does not appear if a role named Greengrass_ServiceRole is in your AWS account.

**Detach the Greengrass Service Role (Console)**

Use the following procedure to detach the Greengrass service role from your AWS account in the AWS Region currently selected in the console. 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. In the AWS IoT console, in the navigation pane, choose Settings.
2. Under Greengrass service role, choose Detach.
3. In the confirmation dialog box, choose Detach role.

**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.

### Managing the Greengrass Service Role (CLI)

In the following procedures, we assume that the AWS CLI is installed and configured to use your AWS account ID. For more information, see Installing the AWS Command Line Interface 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:

- Get Your Greengrass Service Role (p. 446)
- Create the Greengrass Service Role (p. 446)
- Remove the Greengrass Service Role (p. 447)

### 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 greengrass get-service-role-for-account --region region
```

If a Greengrass service role is already associated with your account, the following role metadata is returned.

```json
{
    "AssociatedAt": "timestamp",
    "RoleArn": "arn:aws:iam::account-id:role/path/role-name"
}
```

If no role metadata is returned, 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)

**To create the service role using IAM**

1. Create the 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.
Manage the Service Role (CLI)

Linux, macOS, or Unix

```bash
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"
  }
  ]
}''
```

Windows Command Prompt

```bash
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"}]}
```

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.

```bash
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`).

```bash
aws greengrass associate-service-role-to-account --role-arn role-arn --region region
```

If successful, the following response is returned.

```json
{
  "AssociatedAt": "timestamp"
}
```

Remove the Greengrass Service Role (CLI)

Use the following procedure 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`).

```bash
aws greengrass disassociate-service-role-from-account --region region
```
If successful, the following response is returned.

```json
{
  "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`

**Hardware Security Integration**

This feature is available for AWS IoT Greengrass Core v1.7 and later.

AWS IoT Greengrass supports the use of hardware security modules (HSM) through the PKCS#11 interface (p. 456) for secure storage and offloading of private keys. This prevents keys from being exposed or duplicated in software. Private keys can be securely stored on hardware modules, such as HSMs, Trusted Platform Modules (TPM), or other cryptographic elements.

Search for devices that are qualified for this feature in the *AWS Partner Device Catalog*.

The following diagram shows the hardware security architecture for an AWS IoT Greengrass core.
On a standard installation, AWS IoT Greengrass uses two private keys. One key is used by the AWS IoT client component during the Transport Layer Security (TLS) handshake when a Greengrass core connects to AWS IoT. (This key is also referred to as the core private key.) The other key is used by the local MQTT server, which enables Greengrass devices to communicate with the Greengrass core. If you want to use hardware security for both components, you can use a shared private key or separate private keys. For more information, see the section called "Provisioning Practices" (p. 452).

Note
On a standard installation, the local secrets manager also uses the IoT client key for its encryption process, but you can use your own private key. It must be an RSA key with a minimum length of 2048 bits. For more information, see the section called "Specify the Private Key for Secret Encryption" (p. 265).

Requirements

Before you can configure hardware security for a Greengrass core, you must have the following:

- A hardware security module (HSM) that supports your target private key configuration for the IoT client, local MQTT server, and local secrets manager components. The configuration can include one, two, or three hardware-based private keys, depending on whether you configure the components to share keys. For more information about private key support, see the section called “Security Principals” (p. 439).
  - For RSA keys: An RSA-2048 key size (or larger) and PKCS#1 v1.5 (p. 456) signature scheme.
  - For EC keys: An NIST P-256 or NIST P-384 curve.

Note
Search for devices that are qualified for this feature in the AWS Partner Device Catalog.

- A PKCS#11 provider library that is loadable at runtime (using libdl) and provides PKCS#11 (p. 456) functions.
- The hardware module must be resolvable by slot label, as defined in the PKCS#11 specification.
- The private key must be generated and loaded on the HSM by using the vendor-provided provisioning tools.
- The private key must be resolvable by object label.
- The core device certificate. This is an AWS IoT client certificate that corresponds to the private key.
- If you’re using the Greengrass OTA update agent, the OpenSSL libp11 PKCS#11 wrapper library must be installed. For more information, see the section called “Configure OTA Updates” (p. 454).
In addition, make sure that the following conditions are met:

- The IoT client certificates that are associated with the private key are registered in AWS IoT and activated. You can verify this from the Manage page for the core thing in the AWS IoT console.
- The AWS IoT Greengrass Core software v1.7 or later is installed on the core device, as described in Module 2 (p. 76) of the Getting Started tutorial. Version 1.9 is required to use an EC key for the MQTT server.
- The certificates are attached to the Greengrass core. You can verify this from the Manage page for the core thing in the AWS IoT console.

**Note**
Currently, AWS IoT Greengrass doesn't support loading the CA certificate or AWS IoT client certificate directly from the HSM. The certificates must be loaded as plain-text files on the file system in a location that can be read by Greengrass.

## Hardware Security Configuration for an AWS IoT Greengrass Core

Hardware security is configured in the Greengrass configuration file. This is the config.json (p. 24) file that's located in the /greengrass-root/config directory.

**Note**
To walk through the process of setting up an HSM configuration using a pure software implementation, see the section called “Module 7: Simulating Hardware Security Integration” (p. 142).

**Important**
The simulated configuration in the example doesn’t provide any security benefits. It’s intended to allow you to learn about the PKCS#11 specification and do initial testing of your software if you plan to use a hardware-based HSM in the future.

To configure hardware security in AWS IoT Greengrass, you edit the crypto object in config.json.

When using hardware security, the crypto object is used to specify paths to certificates, private keys, and assets for the PKCS#11 provider library on the core, as shown in the following example.

```json
"crypto": {
  "PKCS11": {
    "OpenSSLengine": "/path-to-p11-openssl-engine",
    "P11Provider": "/path-to-pkcs11-provider-so",
    "slotLabel": "crypto-token-name",
    "slotUserPin": "crypto-token-user-pin"
  },
  "principals": {
    "IoTCertificate": {
      "privateKeyPath": "pkcs11:object=core-private-key-label;type=private",
      "certificatePath": "file:///path-to-core-device-certificate"
    },
    "MQTTServerCertificate": {
      "privateKeyPath": "pkcs11:object=server-private-key-label;type=private"
    },
    "SecretsManager": {
      "privateKeyPath": "pkcs11:object=core-private-key-label;type=private"
    }
  },
  "caPath": "file:///path-to-root-ca"
}
```

The crypto object contains the following properties:
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>caPath</td>
<td>The absolute path to the AWS IoT root CA.</td>
<td>Must be a file URI of the form: file:///absolute/path/to/file. Make sure that your endpoints correspond to your certificate type (p. 45).</td>
</tr>
<tr>
<td>PKCS11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenSSLEngine</td>
<td>Optional. The absolute path to the OpenSSL engine .so file to enable PKCS#11 support on OpenSSL.</td>
<td>Must be a path to a file on the file system.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This property is required if you're using the Greengrass OTA update agent with hardware security. For more information, see the section called “Configure OTA Updates” (p. 454).</td>
</tr>
<tr>
<td>P11Provider</td>
<td>The absolute path to the PKCS#11 implementation’s libdl-loadable library.</td>
<td>Must be a path to a file on the file system.</td>
</tr>
<tr>
<td>slotLabel</td>
<td>The slot label that’s used to identify the hardware module.</td>
<td>Must conform to PKCS#11 label specifications.</td>
</tr>
<tr>
<td>slotUserPin</td>
<td>The user pin that’s used to authenticate the Greengrass core to the module.</td>
<td>Must have sufficient permissions to perform C_Sign with the configured private keys.</td>
</tr>
<tr>
<td>principals</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IoTCertificate</td>
<td>The certificate and private key that the core uses to make requests to AWS IoT.</td>
<td>For file system storage, must be a file URI of the form: file:///absolute/path/to/file. For RFC 7512 PKCS#11 storage, must be an RFC 7512 PKCS#11 path that specifies the object label.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IoTCertificate.privateKeyPath</td>
<td>The path to the core private key.</td>
<td>For file system storage, must be a file URI of the form: file:///absolute/path/to/file. For RFC 7512 PKCS#11 storage, must be an RFC 7512 PKCS#11 path that specifies the object label.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MQTTServerCertificate</td>
<td>Optional. The private key that the core uses in combination with the certificate to act as an MQTT server or gateway.</td>
<td>Use this value to specify your own private key for the local MQTT server.</td>
</tr>
<tr>
<td>MQTTServerCertificate.privateKeyPath</td>
<td>The path to the local MQTT server private key.</td>
<td>Use this value to specify your own private key for the local MQTT server.</td>
</tr>
</tbody>
</table>
Provisioning Practices for AWS IoT Greengrass

Hardware Security

The following are security and performance-related provisioning practices.

Security

- Generate private keys directly on the HSM by using the internal hardware random-number generator.

  **Note**
  If you configure private keys to use with this feature (by following the instructions provided by the hardware vendor), be aware that AWS IoT Greengrass currently supports only the PKCS1 v1.5 padding mechanism for encryption and decryption of local secrets (p. 263). AWS IoT Greengrass doesn't support Optimal Asymmetric Encryption Padding (OAEP).

- Configure private keys to prohibit export.

- Use the provisioning tool that's provided by the hardware vendor to generate a certificate signing request (CSR) using the hardware-protected private key, and then use the AWS IoT console to generate a client certificate.

  **Note**
  The practice of rotating keys doesn't apply when private keys are generated on an HSM.
Performance

The following diagram shows the AWS IoT client component and local MQTT server on the AWS IoT Greengrass core. If you want to use an HSM configuration for both components, you can use the same private key or separate private keys. If you use separate keys, they must be stored in the same slot.

**Note**

AWS IoT Greengrass doesn’t impose any limits on the number of keys that you store on the HSM, so you can store private keys for the IoT client, MQTT server, and secrets manager components. However, some HSM vendors might impose limits on the number of keys you can store in a slot.

In general, the IoT client key is not used very frequently because the AWS IoT Greengrass Core software maintains long-lived connections to the cloud. However, the MQTT server key is used every time that a Greengrass device connects to the core. These interactions directly affect performance.

When the MQTT server key is stored on the HSM, the rate at which devices can connect depends on the number of RSA signature operations per second that the HSM can perform. For example, if the HSM takes 300 milliseconds to perform an RSASSA-PKCS1-v1.5 signature on an RSA-2048 private key, then only three devices can connect to the Greengrass core per second. After the connections are made, the HSM is no longer used and the standard Greengrass limits apply.

To mitigate performance bottlenecks, you can store the private key for the MQTT server on the file system instead of on the HSM. With this configuration, the MQTT server behaves as if hardware security isn’t enabled.

AWS IoT Greengrass supports multiple key-storage configurations for the IoT client and MQTT server components, so you can optimize for your security and performance requirements. The following table includes example configurations.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>IoT Key</th>
<th>MQTT Key</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSM Shared Key</td>
<td>HSM: Key A</td>
<td>HSM: Key A</td>
<td>Limited by the HSM or CPU</td>
</tr>
<tr>
<td>HSM Separate Keys</td>
<td>HSM: Key A</td>
<td>HSM: Key B</td>
<td>Limited by the HSM or CPU</td>
</tr>
<tr>
<td>HSM for IoT only</td>
<td>HSM: Key A</td>
<td>File System: Key B</td>
<td>Limited by the CPU</td>
</tr>
</tbody>
</table>
Supported Cipher Suites for Hardware Security Integration

AWS IoT Greengrass supports a set of cipher suites when the core is configured for hardware security. This is a subset of the cipher suites that are supported when the core is configured to use file-based security. For more information, see the section called “TLS Cipher Suites Support” (p. 442).

**Note**
When connecting to the Greengrass core from Greengrass devices over the local network, be sure to use one of the supported cipher suites to make the TLS connection.

Configure Support for Over-the-Air Updates

To enable over-the-air (OTA) updates of the AWS IoT Greengrass Core software when using hardware security, you must install the OpenSC libp11 PKCS#11 wrapper library and edit the Greengrass configuration file. For more information about OTA updates, see OTA Updates of AWS IoT Greengrass Core Software (p. 148).

1. Stop the AWS Greengrass daemon.

   ```
   cd /greengrass-root/ggc/core/
   sudo ./greengrassd stop
   ```

   **Note**
   `greengrass-root` represents the path where the AWS IoT Greengrass Core software is installed on your device. If you installed the software by following the steps in the Getting Started (p. 64) tutorial, then this is the `/greengrass` directory.
2. Install the OpenSSL engine.
   
   ```
   sudo apt-get install libengine-pkcs11-openssl
   ```

3. Find the path to the OpenSSL engine (libpkcs11.so) on your system:
   a. Get the list of installed packages for the library.
      ```
      sudo dpkg -L libengine-pkcs11-openssl
      ```
      The libpkcs11.so file is located in the engines directory.
   b. Copy the full path to the file (for example, /usr/lib/ssl/engines/libpkcs11.so).

4. Open the Greengrass configuration file. This is the `config.json` (p. 24) file in the `/greengrass-root/config` directory.

5. For the `OpenSSLEngine` property, enter the path to the libpkcs11.so file.
   ```
   {  
     "crypto": {  
       "caPath" : "file:///path-to-root-ca",  
       "PKCS11" : {  
         "OpenSSLEngine" : "/path-to-pkcs11-openssl-engine",  
         "P11Provider" : "/path-to-pkcs11-provider-so",  
         "slotLabel" : "crypto-token-name",  
         "slotUserPin" : "crypto-token-user-pin"  
       },  
       ...  
     }  
   }  
   ```

   **Note**
   If the `OpenSSLEngine` property doesn't exist in the PKCS11 object, then add it.

6. Start the AWS Greengrass daemon.
   ```
   cd /greengrass-root/ggc/core/
   sudo ./greengrassd start
   ```

---

**Backward Compatibility with Earlier Versions of the AWS IoT Greengrass Core Software**

The AWS IoT Greengrass Core software with hardware security support is fully backward compatible with `config.json` files that are generated for v1.6 and earlier. If the `crypto` object is not present in the `config.json` configuration file, then AWS IoT Greengrass uses the file-based `coreThing.certPath`, `coreThing.keyPath`, and `coreThing.caPath` properties. This backward compatibility applies to Greengrass OTA updates, which do not overwrite a file-based configuration that's specified in `config.json`.

**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, you should contact your AWS Enterprise Support representative with integration-related questions.

See Also

- RFC 7512
- PKCS #1: RSA Encryption Version 1.5
Monitoring with 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 and to the local file system of your core device. 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.

Logging is configured at the group level. For steps that show how to configure logging for an AWS IoT Greengrass group, see the section called “Configure Logging for AWS IoT Greengrass” (p. 459).

CloudWatch Logs

If you configure CloudWatch logging, you can view the logs on the Logs page of the Amazon CloudWatch console. Log groups for AWS IoT Greengrass logs use the following naming conventions:

/aws/greengrass/GreengrassSystem/greengrass-system-component-name
/aws/greengrass/Lambda/aws-region/account-id/lambda-function-name

Each log group contains log streams that use the following naming convention:

date/account-id/greengrass-group-id/name-of-core-that-generated-log

The following considerations apply when you use CloudWatch Logs:

* Your Greengrass group role must allow AWS IoT Greengrass to write to CloudWatch Logs. To grant permissions, embed the following inline policy in your group role.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Resource": ["arn:aws:logs:*:*:*"]
        }
    ]
}
```

You can grant more granular access to your log resources. For more information, see Using Identity-Based Policies (IAM Policies) for CloudWatch Logs in the Amazon CloudWatch User Guide.
File System Logs

If you configure file system logging, the log files are stored under greengrass-root/ggc/var/log on the core device. The following is the high-level directory structure:

```plaintext
greengrass-root/ggc/var/log
  - crash.log
  - system
    - log files for each Greengrass system component
  - user
    - log files generated by each user-defined Lambda function
```

**Note**
By default, greengrass-root is the /greengrass directory. If a write directory (p. 52) is configured, then the logs are under that directory.

The following considerations apply when you use file system logs:

- Reading AWS IoT Greengrass logs on the file system requires root permissions.
- AWS IoT Greengrass supports size-based rotation and automatic cleanup when the amount of log data is close to the configured limit.
- The crash.log file is available in file system logs only. This log isn't written to CloudWatch Logs.
- Disk usage limitations apply. For more information, see the section called “Logging Limitations” (p. 461).

**Note**
Logs for AWS IoT Greengrass Core software v1.0 are stored under the greengrass-root/var/log directory.

Default Logging Configuration

If logging settings aren't explicitly configured, AWS IoT Greengrass uses the following default logging configuration after the first group deployment.

**AWS IoT Greengrass System Components**
- Type - FileSystem
- Component - GreengrassSystem
- Level - INFO
- Space - 128 KB

**User-defined Lambda Functions**
- Type - FileSystem
- Component - Lambda
- Level - INFO
Configure Logging for AWS IoT Greengrass

You can use the AWS IoT console or the AWS IoT Greengrass APIs (p. 459) to configure AWS IoT Greengrass logging.

**Note**
To allow AWS IoT Greengrass to write logs to CloudWatch Logs, your group role must allow the required CloudWatch Logs actions (p. 457).

### Configure Logging (Console)

You can configure logging on the group's **Settings** page.

1. In the AWS IoT console, choose **Greengrass**, and then choose **Groups**.
2. Choose the group where you want to configure logging.
3. On the group configuration page, choose **Settings**.
4. Choose the logging location, as follows:
   - To configure CloudWatch logging, for **CloudWatch logs configuration**, choose **Edit**.
   - To configure file system logging, for **Local logs configuration**, choose **Edit**.

   You can configure logging for one location or both locations.

5. On the **Configure Group logging** page, choose **Add another log type**.
6. Choose the event source, as follows:
   - To log events from user-defined Lambda functions, choose **User Lambdas**.
   - To log events from AWS IoT Greengrass system components, choose **Greengrass system**.

   You can choose one component or both components.

7. Choose **Update**.
8. Choose the lowest level of events that you want to log. Events below this threshold are filtered out and aren't stored.
9. For file system logs, specify a disk space limit.
10. Choose **Save**.

### Configure Logging (API)

You can use AWS IoT Greengrass logger APIs to configure logging programmatically. For example, use the **CreateLoggerDefinition** action to create a logger definition based on a **LoggerDefinitionVersion** payload, which uses the following syntax:

```
{
  "Loggers": [
```
LoggerDefinitionVersion is an array of one or more Logger objects that have the following properties:

- **Id**
  - An identifier for the logger.

- **Type**
  - The storage mechanism for log events. When AWSCloudWatch is used, log events are sent to CloudWatch Logs. When FileSystem is used, log events are stored on the local file system.

  Valid values: AWSCloudWatch, FileSystem

- **Component**
  - The source of the log event. When GreengrassSystem is used, events from Greengrass system components are logged. When Lambda is used, events from user-defined Lambda functions are logged.

  Valid values: GreengrassSystem, Lambda

- **Level**
  - The log-level threshold. Log events below this threshold are filtered out and aren't stored.

  Valid values: DEBUG, INFO (recommended), WARN, ERROR, FATAL

- **Space**
  - The maximum amount of local storage, in KB, to use for storing logs. This field applies only when Type is set to FileSystem.

### Configuration Example

The following LoggerDefinitionVersion example specifies a logging configuration that:

- Turns on file system ERROR (and above) logging for AWS IoT Greengrass system components.
- Turns on file system INFO (and above) logging for user-defined Lambda functions.
- Turns on CloudWatch INFO (and above) logging for user-defined Lambda functions.

```json
{
    "Name": "LoggingExample",
    "InitialVersion": {
        "Loggers": [
            {
                "Id": "string",
                "Type": "FileSystem|AWSCloudWatch",
                "Component": "GreengrassSystem|Lambda",
                "Level": "DEBUG|INFO|WARN|ERROR|FATAL",
                "Space": "integer"
            },
            {
                "Id": "string",
                ...
            }
        ]
    }
}
```
After you create a logger definition version, you can use its version ARN to create a group version before deploying the group.

# Logging Limitations

AWS IoT Greengrass has the following logging limitations.

## Transactions per Second

When logging to CloudWatch is enabled, the logging component batches log events locally before sending them to CloudWatch, so you can log at a rate higher than five requests per second per log stream.

## Memory

If AWS IoT Greengrass is configured to send logs to CloudWatch and a Lambda function logs more than 5 MB/second for a prolonged period of time, the internal processing pipeline eventually fills up. The theoretical worst case is 6 MB per Lambda function.

## Clock Skew

When logging to CloudWatch is enabled, the logging component signs requests to CloudWatch using the normal Signature Version 4 signing process. If the system time on the AWS IoT Greengrass core device is out of sync by more than 15 minutes, then the requests are rejected.

## Disk Usage

Use the following formula to calculate the total maximum amount of disk usage for logging:

\[
\text{greengrass-system-component-space} \times 8 \quad // 7 \text{ if automatic IP detection is disabled} \\
+ 128\text{KB} \quad // \text{the internal log for the local logging}
\]
Log Loss

If your AWS IoT Greengrass core device is configured to log only to CloudWatch and there’s no internet connectivity, you have no way to retrieve the logs currently in the memory.

When Lambda functions are terminated (for example, during deployment), a few seconds’ worth of logs are not written to CloudWatch.

Logging AWS IoT Greengrass API Calls with AWS CloudTrail

AWS IoT Greengrass 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. CloudTrail captures all API calls for AWS IoT Greengrass as events. The calls 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 Amazon 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.

To learn more about CloudTrail, see the AWS CloudTrail User Guide.

AWS IoT Greengrass 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 Amazon 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 Amazon 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
Understanding AWS IoT Greengrass Log File Entries

All AWS IoT Greengrass actions are logged by CloudTrail and are documented in the AWS IoT Greengrass API Reference. For example, calls to the `AssociateServiceRoleToAccount`, `GetGroupVersion`, `GetConnectivityInfo`, and `CreateFunctionDefinition` 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.

Understanding AWS IoT Greengrass Log File Entries

A trail is a configuration that enables delivery of events as log files to an Amazon S3 bucket that you specify. CloudTrail log files contain one or more log entries. An event represents a single request from any source and 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 ` AssociateServiceRoleToAccount` action.

```json
{
    "eventVersion": "1.05",
    "userIdentity": {
        "type": "IAMUser",
        "principalId": "AIDACKCEVSQ6C2EXAMPLE",
        "arn": "arn:aws:iam::123456789012:user/Mary_Major",
        "accountId": "123456789012",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "Mary_Major"
    },
    "eventTime": "2018-10-17T17:04:02Z",
    "eventSource": "greengrass.amazonaws.com",
    "eventName": "AssociateServiceRoleToAccount",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "203.0.113.12",
    "userAgent": "ammanager.amazonaws.com",
    "errorCode": "BadRequestException",
    "requestParameters": null,
    "responseElements": {
        "Message": "That role ARN is invalid."
    },
    "requestID": "a5990ec6-d22e-11e8-8ae5-c7d2eEXAMPLE",
    "eventID": "b9070ce5-0238-451a-a9db-2dbf1EXAMPLE",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "recipientAccountId": "123456789012"
}
```
The following example shows a CloudTrail log entry that demonstrates the `GetGroupVersion` action.

```
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "IAMUser",
    "principalId": "AIDACKCEVSQ6C2EXAMPLE",
    "arn": "arn:aws:iam::123456789012:user/Mary_Major",
    "accountId": "123456789012",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
    "userName": "Mary_Major",
    "sessionContext": {
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2018-10-17T18:14:57Z"
      }
    },
    "invokedBy": "apimanager.amazonaws.com"
  },
  "eventTime": "2018-10-17T18:15:11Z",
  "eventSource": "greengrass.amazonaws.com",
  "eventName": "GetGroupVersion",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "203.0.113.12",
  "userAgent": "apimanager.amazonaws.com",
  "requestParameters": {
    "GroupVersionId": "6c477753-dbf2-4cb8-acc3-5ba4eEXAMPLE",
    "GroupId": "90fcf6d6-413c-4515-93a8-00056EXAMPLE"
  },
  "responseElements": null,
  "requestID": "95dcffce-d238-11e8-9240-a3993EXAMPLE",
  "eventID": "8a608034-82ed-431b-b5e0-87fbdEXAMPLE",
  "readOnly": true,
  "eventType": "AwsApiCall",
  "recipientAccountId": "123456789012"
}
```

The following example shows a CloudTrail log entry that demonstrates the `GetConnectivityInfo` action.

```
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "IAMUser",
    "principalId": "AIDACKCEVSQ6C2EXAMPLE",
    "arn": "arn:aws:iam::123456789012:user/Mary_Major",
    "accountId": "123456789012",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
    "userName": "Mary_Major"
  },
  "eventTime": "2018-10-17T18:15:11Z",
  "eventSource": "greengrass.amazonaws.com",
  "eventName": "GetConnectivityInfo",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "203.0.113.12",
  "userAgent": "apimanager.amazonaws.com",
  "requestParameters": {
    "ThingName": "us-east-1_CIS_1539795000000_"
  },
  "responseElements": null,
  "requestID": "63e3ebe3-d22e-11e8-9dd5-5ba4fEXAMPLE",
  "eventID": "db2260d1-a8cc-4a65-b92a-13f65EXAMPLE",
  "readOnly": true,
  "eventType": "AwsApiCall"
}
```
The following example shows a CloudTrail log entry that demonstrates the `CreateFunctionDefinition` action.

```
{
    "eventVersion": "1.05",
    "userIdentity": {
        "type": "IAMUser",
        "principalId": "AIDACKCEVSQ6C2EXAMPLE",
        "arn": "arn:aws:iam::123456789012:user/Mary_Major",
        "accountId": "123456789012",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "Mary_Major"
    },
    "eventTime": "2018-10-17T18:01:11Z",
    "eventSource": "greengrass.amazonaws.com",
    "eventName": "CreateFunctionDefinition",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "203.0.113.12",
    "userAgent": "apimanager.amazonaws.com",
    "requestParameters": {
        "InitialVersion": "***"
    },
    "responseElements": {
        "CreationTimestamp": "2018-10-17T18:01:11.449Z",
        "LatestVersion": "dae06a61-c32c-41e9-b983-ee5cfEXAMPLE",
        "LatestVersionArn": "arn:aws:greengrass:us-east-1:123456789012:/greengrass/definition/functions/7a94847d-d4d2-406c-9796-a3529EXAMPLE/versions/dae06a61-c32c-41e9-b983-ee5cfEXAMPLE",
        "LastUpdatedTimestamp": "2018-10-17T18:01:11.449Z",
        "Id": "7a94847d-d4d2-406c-9796-a3529EXAMPLE",
        "Arn": "arn:aws:greengrass:us-east-1:123456789012:/greengrass/definition/functions/7a94847d-d4d2-406c-9796-a3529EXAMPLE"
    },
    "requestID": "a17d4b96-d236-11e8-a74e-3db27EXAMPLE",
    "eventType": "AwsApiCall",
    "recipientAccountId": "123456789012"
}
```

See Also

- Creating an EventBridge Rule That Triggers on an AWS API Call Using CloudTrail in the Amazon EventBridge User Guide
- AWS IoT Greengrass API Reference

CloudTrail Logs

AWS IoT Greengrass 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. For more information, see the section called "Logging AWS IoT Greengrass API Calls with AWS CloudTrail" (p. 462).
Tagging Your AWS IoT Greengrass Resources

Tags can help you organize and manage your AWS IoT Greengrass groups. You can use tags to assign metadata to groups, bulk deployments, and the cores, devices, and other resources that are added to groups. Tags can also be used in IAM policies to define conditional access to your Greengrass resources.

**Note**
Currently, Greengrass resource tags are not supported for AWS IoT billing groups or cost allocation reports.

Tag Basics

Tags allow you to categorize your AWS IoT Greengrass resources, for example, by purpose, owner, and environment. When you have many resources of the same type, you can quickly identify a resource based on the tags that are attached to it. A tag consists of a key and optional value, both of which you define. We recommend that you design a set of tag keys for each resource type. Using a consistent set of tag keys makes it easier for you to manage your resources. For example, you can define a set of tags for your groups that helps you track the factory location of your core devices. For more information, see AWS Tagging Strategies.

Tagging Support in the AWS IoT Console

You can create, view, and manage tags for your Greengrass Group resources in the AWS IoT console. Before you create tags, be aware of tagging restrictions. For more information, see Tag Naming and Usage Conventions in the Amazon Web Services General Reference.

To assign tags when you create a group

You can assign tags to a group when you create the group. To show the tagging input fields, on the Name your Group dialog box, choose Apply tags to the Group (optional).

![Image of tagging input fields](image)
To view and manage tags from the group configuration page

You can view and manage tags from the group configuration page. On the **Tags** page for the group, choose **Add tags** or **Manage tags** to add, edit, or remove group tags.

### Tagging Support in the AWS IoT Greengrass API

You can use the AWS IoT Greengrass API to create, list, and manage tags for AWS IoT Greengrass resources that support tagging. Before you create tags, be aware of tagging restrictions. For more information, see [Tag Naming and Usage Conventions](#) in the *Amazon Web Services General Reference*.

- To add tags during resource creation, define them in the `tags` property of the resource.
- To add tags after a resource is created, or to update tag values, use the `TagResource` action.
- To remove tags from a resource, use the `UntagResource` action.
- To retrieve the tags that are associated with a resource, use the `ListTagsForResource` action or get the resource and inspect its `tags` property.

The following table lists resources you can tag in the AWS IoT Greengrass API and their corresponding **Create** and **Get** actions.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Create</th>
<th>Get</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group</td>
<td><code>CreateGroup</code></td>
<td><code>GetGroup</code></td>
</tr>
<tr>
<td>ConnectorDefinition</td>
<td><code>CreateConnectorDefinition</code></td>
<td><code>GetConnectorDefinition</code></td>
</tr>
<tr>
<td>CoreDefinition</td>
<td><code>CreateCoreDefinition</code></td>
<td><code>GetCoreDefinition</code></td>
</tr>
<tr>
<td>DeviceDefinition</td>
<td><code>CreateDeviceDefinition</code></td>
<td><code>GetDeviceDefinition</code></td>
</tr>
<tr>
<td>FunctionDefinition</td>
<td><code>CreateFunctionDefinition</code></td>
<td><code>GetFunctionDefinition</code></td>
</tr>
<tr>
<td>LoggerDefinition</td>
<td><code>CreateLoggerDefinition</code></td>
<td><code>GetLoggerDefinition</code></td>
</tr>
<tr>
<td>ResourceDefinition</td>
<td><code>CreateResourceDefinition</code></td>
<td><code>GetResourceDefinition</code></td>
</tr>
<tr>
<td>SubscriptionDefinition</td>
<td><code>CreateSubscriptionDefinition</code></td>
<td><code>GetSubscriptionDefinition</code></td>
</tr>
<tr>
<td>BulkDeployment</td>
<td><code>StartBulkDeployment</code></td>
<td><code>GetBulkDeploymentStatus</code></td>
</tr>
</tbody>
</table>
Use the following actions to list and manage tags for resources that support tagging:

- **TagResource**: Adds tags to a resource. Also used to change the value of the tag's key-value pair.
- **ListTagsForResource**: Lists the tags for a resource.
- **UntagResource**: Removes tags from a resource.

You can add or remove tags on 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 overwrites the old 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 the resource are also deleted.

**Note**

Don't confuse resource tags with the attributes that you can assign to AWS IoT things. Although Greengrass cores are AWS IoT things, the resource tags that are described in this topic are attached to a `CoreDefinition`, not the core thing.

## 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. You can tag resources during creation (called *tag on create*) so you don't have to run custom tagging scripts later. When new environments are launched with tags, the corresponding IAM permissions are applied automatically.

The following condition context keys and values can be used in the `Condition` element (also called the `Condition block`) of the policy.

- **greengrass:ResourceTag/tag-key**: `tag-value`
  
  Allow or deny user actions on resources with specific tags.

- **aws:RequestTag/tag-key**: `tag-value`
  
  Require that a specific tag be used (or not used) when making API requests to create or modify tags on a taggable resource.

- **aws:TagKeys**: `[tag-key, ...]`

  Require that a specific set of tag keys be used (or not used) when making an API request to create or modify a taggable resource.

Condition context keys and values can be used only on AWS IoT Greengrass actions that act on a taggable resource. These actions take the resource as a required parameter. For example, you can set conditional access on the `GetGroupVersion`. You can't set conditional access on `AssociateServiceRoleToAccount` because no taggable resource (for example, group, core definition, or device definition) is referenced in the request.

For more information, see [Controlling Access Using Tags](#) and [IAM JSON Policy Reference](#) in the *IAM User Guide*. The JSON policy reference includes detailed syntax, descriptions and examples of the elements, variables, and evaluation logic of JSON policies in IAM.

## Example IAM Policies

The following example policy applies tag-based permissions that constrain a beta user to actions on beta resources only.

- The first statement allows an IAM user to act on resources that have the `env=beta` tag only.
• The second statement prevents an IAM user from removing the env=beta tag from resources. This protects the user from removing their own access.

**Note**

If you use tags to control access to resources, you should also manage the permissions that allow users to add tags or remove tags from those same resources. Otherwise, in some cases, it might be possible for users to circumvent your restrictions and gain access to a resource by modifying its tags.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "greengrass:*",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "greengrass:ResourceTag/env": "beta"
                }
            }
        },
        {
            "Effect": "Deny",
            "Action": "greengrass:UntagResource",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/env": "beta"
                }
            }
        }
    ]
}
```

To allow users to tag on create, you must give them appropriate permissions. The following example policy includes the "aws:RequestTag/env": "beta" condition on the greengrass:TagResource and greengrass:CreateGroup actions, which allows users to create a group only if they tag the group with env=beta. This effectively forces users to tag new groups.

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "greengrass:TagResource",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "aws:RequestTag/env": "beta"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": "greengrass:CreateGroup",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "aws:RequestTag/env": "beta"
                }
            }
        }
    ]
}
```
The following snippet shows how you can specify multiple tag values for a tag key by enclosing them in a list:

```
"StringEquals" : {
    "greengrass:ResourceTag/env" : ["dev", "test"]
}
```

See Also

- Tagging AWS Resources in the *Amazon Web Services General Reference*
AWS CloudFormation Support for AWS IoT Greengrass

AWS CloudFormation is a service that can help you create, manage, and replicate your AWS resources. You can use AWS CloudFormation templates to define AWS IoT Greengrass groups and the devices, subscriptions, and other components that you want to deploy. For an example, see the section called “Example Template” (p. 472).

The resources and infrastructure that you generate from a template is called a stack. You can define all of your resources in one template or refer to resources from other stacks. For more information about AWS CloudFormation templates and features, see What Is AWS CloudFormation? in the AWS CloudFormation User Guide.

Creating Resources

AWS CloudFormation templates are JSON or YAML documents that describe the properties and relationships of AWS resources. The following AWS IoT Greengrass resources are supported:

- Groups
- Cores
- Devices
- Lambda functions
- Connectors
- Resources (local, machine learning, and secret)
- Subscriptions
- Loggers (logging configurations)

In AWS CloudFormation templates, the structure and syntax of Greengrass resources are based on the AWS IoT Greengrass API. For example, the example template (p. 472) associates a top-level DeviceDefinition with a DeviceDefinitionVersion that contains an individual device. For more information, see the section called “Overview of the Group Object Model” (p. 157).

The AWS IoT Greengrass Resource Types Reference in the AWS CloudFormation User Guide describes the Greengrass resources that you can manage with AWS CloudFormation. When you use AWS CloudFormation templates to create Greengrass resources, we recommend that you manage them only from AWS CloudFormation. For example, you should update your template if you want to add, change, or remove a device (instead of using the AWS IoT Greengrass API or AWS IoT console). This allows you to use rollback and other AWS CloudFormation change management features. For more information about using AWS CloudFormation to create and manage your resources and stacks, see Working with Stacks in the AWS CloudFormation User Guide.

For a walkthrough that shows how to create and deploy AWS IoT Greengrass resources in an AWS CloudFormation template, see Automating AWS IoT Greengrass Setup with AWS CloudFormation on The Internet of Things on AWS Official Blog.
Deploying Resources

After you create an AWS CloudFormation stack that contains your group version, you can use the AWS CLI or AWS IoT console to deploy it.

**Note**
To deploy a group, you must have a Greengrass service role associated with your AWS account. The service role allows AWS IoT Greengrass to access your resources in AWS Lambda and other AWS services. This role should exist if you already deployed a Greengrass group in the current AWS Region. For more information, see the section called "Greengrass Service Role" (p. 443).

**To deploy the group (AWS CLI)**
- Run the `create-deployment` command.

```bash
aws greengrass create-deployment --group-id GroupId --group-version-id GroupVersionId --deployment-type NewDeployment
```

**Note**
The `CommandToDeployGroup` statement in the example template (p. 472) shows how to output the command with your group and group version IDs when you create a stack.

**To deploy the group (console)**
1. In the AWS IoT console, choose **Greengrass**, and then choose **Groups**.
2. Choose your group.
3. On the group configuration page, from **Actions**, choose **Deploy**.

This deploys the group configuration to your AWS IoT Greengrass core device. For troubleshooting help, see *Troubleshooting* (p. 516).

**Example Template**

The following example template creates a Greengrass group that contains a core, device, function, logger, subscription, and two resources. To do this, the template follows the object model of the AWS IoT Greengrass API. For example, the devices that you want to add to the group are contained in a `DeviceDefinitionVersion` resource, which is associated with a `DeviceDefinition` resource. To add the devices to the group, the group version references the ARN of the `DeviceDefinitionVersion`.

The template includes parameters that let you specify the certificate ARNs for the core and device and the version ARN of the source Lambda function (which is an AWS Lambda resource). It uses the `Ref` and `GetAtt` intrinsic functions to reference IDs, ARNs, and other attributes that are required to create Greengrass resources.

The template also defines two AWS IoT devices (things), which represent the core and device that are added to the Greengrass group.
After you create the stack with your Greengrass resources, you can use the AWS CLI or the AWS IoT console to deploy the group (p. 472).

**Note**
The `CommandToDeployGroup` statement in the example shows how to output a complete `create-deployment` CLI command that you can use to deploy your group.

**JSON**

```json
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Description": "AWS IoT Greengrass example template that creates a group version with a core, device, function, logger, subscription, and resources.",
    "Parameters": {
        "CoreCertificateArn": {
            "Type": "String"
        },
        "DeviceCertificateArn": {
            "Type": "String"
        },
        "LambdaVersionArn": {
            "Type": "String"
        }
    },
    "Resources": {
        "TestCore1": {
            "Type": "AWS::IoT::Thing",
            "Properties": {
                "ThingName": "TestCore1"
            }
        },
        "TestCoreDefinition": {
            "Type": "AWS::Greengrass::CoreDefinition",
            "Properties": {
                "Name": "DemoTestCoreDefinition"
            }
        },
        "TestCoreDefinitionVersion": {
            "Type": "AWS::Greengrass::CoreDefinitionVersion",
            "Properties": {
                "CoreDefinitionId": {
                    "Ref": "TestCoreDefinition"
                },
                "Cores": [
                    {
                        "Id": "TestCore1",
                        "CertificateArn": {
                            "Ref": "CoreCertificateArn"
                        },
                        "SyncShadow": "false",
                        "ThingArn": {
                            "Fn::Join": [
                                "",
                                [
                                    "arn:aws:iot",
                                    {
                                        "Ref": "AWS::Region"
                                    },
                                    {
                                        "Ref": "AWS::AccountId"
                                    },
                                    "thing/TestCore1"
                                ]
                            ]
                        }
                    }
                ]
            }
        }
    }
}
```
"TestDevice1": {
  "Type": "AWS::IoT::Thing",
  "Properties": {
    "ThingName": "TestDevice1"
  }
},
"TestDeviceDefinition": {
  "Type": "AWS::Greengrass::DeviceDefinition",
  "Properties": {
    "Name": "DemoTestDeviceDefinition"
  }
},
"TestDeviceDefinitionVersion": {
  "Type": "AWS::Greengrass::DeviceDefinitionVersion",
  "Properties": {
    "DeviceDefinitionId": {
      "Fn::GetAtt": [
        "TestDeviceDefinition",
        "Id"
      ]
    },
    "Devices": [
      {
        "Id": "TestDevice1",
        "CertificateArn": {
          "Ref": "DeviceCertificateArn"
        },
        "SyncShadow": "true",
        "ThingArn": {
          "Fn::Join": ["", [
            "arn:aws:iot",
            {"Ref": "AWS::Region"},
            {"Ref": "AWS::AccountId"},
            "thing/TestDevice1"
          ]]
        }
      }
    ]
  }
},
"TestFunctionDefinition": {
  "Type": "AWS::Greengrass::FunctionDefinition",
  "Properties": {
    "Name": "DemoTestFunctionDefinition"
  }
},
"TestFunctionDefinitionVersion": {
  "Type": "AWS::Greengrass::FunctionDefinitionVersion",
  "Properties": {
    "FunctionDefinitionId": {
      "Fn::GetAtt": [
        "TestFunctionDefinition",
        "Id"
      ]
    }
  }
}
"DefaultConfig": {  
  "Execution": {  
    "IsolationMode": "GreengrassContainer"  
  },  
  "Functions": [  
    {  
      "Id": "TestLambda1",  
      "FunctionArn": {  
        "Ref": "LambdaVersionArn"  
      },  
      "FunctionConfiguration": {  
        "Pinned": "true",  
        "Executable": "run.exe",  
        "ExecArgs": "argument1",  
        "MemorySize": "512",  
        "Timeout": "2000",  
        "EncodingType": "binary",  
        "Environment": {  
          "Variables": {  
            "variable1": "value1"  
          },  
          "ResourceAccessPolicies": [  
            {  
              "ResourceId": "ResourceId1",  
              "Permission": "ro"  
            },  
            {  
              "ResourceId": "ResourceId2",  
              "Permission": "rw"  
            }  
          ],  
          "AccessSysfs": "false",  
          "Execution": {  
            "IsolationMode": "GreengrassContainer",  
            "RunAs": {  
              "Uid": "1",  
              "Gid": "10"  
            }  
          }  
        }  
      }  
    }  
  ],  
  "TestLoggerDefinition": {  
    "Type": "AWS::Greengrass::LoggerDefinition",  
    "Properties": {  
      "Name": "DemoTestLoggerDefinition"  
    }  
  },  
  "TestLoggerDefinitionVersion": {  
    "Type": "AWS::Greengrass::LoggerDefinitionVersion",  
    "Properties": {  
      "LoggerDefinitionId": {  
        "Ref": "TestLoggerDefinition"  
      },  
      "Loggers": [  
        {  
          "Id": "TestLogger1",  
          "Type": "AWSCloudWatch",  
          "Component": "GreengrassSystem",  
          "Level": "INFO"  
        }  
      ]  
    }  
  }  
}


AWS IoT Greengrass Developer Guide
Example Template

"TestResourceDefinition": {
  "Type": "AWS::Greengrass::ResourceDefinition",
  "Properties": {
    "Name": "DemoTestResourceDefinition"
  }
},

"TestResourceDefinitionVersion": {
  "Type": "AWS::Greengrass::ResourceDefinitionVersion",
  "Properties": {
    "ResourceDefinitionId": {
      "Ref": "TestResourceDefinition"
    },
    "Resources": [
      {
        "Id": "ResourceId1",
        "Name": "LocalDeviceResource",
        "ResourceDataContainer": {
          "LocalDeviceResourceData": {
            "SourcePath": "/dev/TestSourcePath1",
            "GroupOwnerSetting": {
              "AutoAddGroupOwner": "false",
              "GroupOwner": "TestOwner"
            }
          }
        }
      },
      {
        "Id": "ResourceId2",
        "Name": "LocalVolumeResourceData",
        "ResourceDataContainer": {
          "LocalVolumeResourceData": {
            "SourcePath": "/dev/TestSourcePath2",
            "DestinationPath": "/volumes/TestDestinationPath2",
            "GroupOwnerSetting": {
              "AutoAddGroupOwner": "false",
              "GroupOwner": "TestOwner"
            }
          }
        }
      }
    ]
  }
},

"TestSubscriptionDefinition": {
  "Type": "AWS::Greengrass::SubscriptionDefinition",
  "Properties": {
    "Name": "DemoTestSubscriptionDefinition"
  }
},

"TestSubscriptionDefinitionVersion": {
  "Type": "AWS::Greengrass::SubscriptionDefinitionVersion",
  "Properties": {
    "SubscriptionDefinitionId": {
      "Ref": "TestSubscriptionDefinition"
    },
    "Subscriptions": [
      {
        "Id": "TestSubscription1",
        "Source": {
          "Fn::Join": [
            ":",
            ["arn:aws:iot",
            ]
          ]
        }
      }
    ]
  }
}
"Ref": "AWS::Region"
},
{
"Ref": "AWS::AccountId"
},
"thing/TestDevice1"
"
"Subject": "TestSubjectUpdated",
"Target": {
"Ref": "LambdaVersionArn"
}
"
"
"TestGroup": {
"Type": "AWS::Greengrass::Group",
"Properties": {
"Name": "DemoTestGroupNewName",
"RoleArn": {
"Fn::Join": [
"::",
[ "arn:aws:iam:",
{ "Ref": "AWS::AccountId" },
"role/TestUser" ]
],
"InitialVersion": {
"CoreDefinitionVersionArn": {
"Ref": "TestCoreDefinitionVersion"
},
"DeviceDefinitionVersionArn": {
"Ref": "TestDeviceDefinitionVersion"
},
"FunctionDefinitionVersionArn": {
"Ref": "TestFunctionDefinitionVersion"
},
"SubscriptionDefinitionVersionArn": {
"Ref": "TestSubscriptionDefinitionVersion"
},
"LoggerDefinitionVersionArn": {
"Ref": "TestLoggerDefinitionVersion"
},
"ResourceDefinitionVersionArn": {
"Ref": "TestResourceDefinitionVersion"
}

"CommandToDeployGroup": {
"Value": {
"Fn::Join": [
" "",
[ "groupVersion=$(cut -d'/' -f6 <<<",
{ "Fn::GetAtt": [
"TestGroup",
"""
"

477
YAML

AWSTemplateFormatVersion: 2010-09-09
Description: >-
AWS IoT Greengrass example template that creates a group version with a core, device, function, logger, subscription, and resources.
parameters:
  CoreCertificateArn:
    Type: String
  DeviceCertificateArn:
    Type: String
  LambdaVersionArn:
    Type: String
Resources:
  TestCore1:
    Type: 'AWS::IoT::Thing'
    Properties:
      ThingName: TestCore1
  TestCoreDefinition:
    Type: 'AWS::Greengrass::CoreDefinition'
    Properties:
      Name: DemoTestCoreDefinition
  TestCoreDefinitionVersion:
    Type: 'AWS::Greengrass::CoreDefinitionVersion'
    Properties:
      CoreDefinitionId: !Ref TestCoreDefinition
      Cores:
      - Id: TestCore1
        CertificateArn: !Ref CoreCertificateArn
        SyncShadow: 'false'
        ThingArn: !Join
          - ':'
          - - 'arn:aws:iot'
          - !Ref 'AWS::Region'
          - !Ref 'AWS::AccountId'
          - thing/TestCore1
  TestDevice1:
    Type: 'AWS::IoT::Thing'
    Properties:
      ThingName: TestDevice1
  TestDeviceDefinition:
    Type: 'AWS::Greengrass::DeviceDefinition'
    Properties:
      Name: DemoTestDeviceDefinition

"LatestVersionArn"
];
"aws --region",
{
  "Ref": "AWS::Region"
},
"greengrass create-deployment --group-id",
{
  "Ref": "TestGroup"
},
"--deployment-type NewDeployment --group-version-id",
"#$groupVersion"
]"
Example Template

TestDeviceDefinitionVersion:
  Type: 'AWS::Greengrass::DeviceDefinitionVersion'
  Properties:
    DeviceDefinitionId: !GetAtt
    - TestDeviceDefinition
    - Id
    Devices:
      - Id: TestDevice1
        CertificateArn: !Ref DeviceCertificateArn
        SyncShadow: 'true'
        ThingArn: !Join
          - ':'
          - !Ref 'AWS::Region'
          - !Ref 'AWS::AccountId'
          - thing/TestDevice1
    TestFunctionDefinition:
      Type: 'AWS::Greengrass::FunctionDefinition'
      Properties:
        Name: DemoTestFunctionDefinition
    TestFunctionDefinitionVersion:
      Type: 'AWS::Greengrass::FunctionDefinitionVersion'
      Properties:
        FunctionDefinitionId: !GetAtt
        - TestFunctionDefinition
        - Id
        DefaultConfig:
          Execution:
            IsolationMode: GreengrassContainer
          Functions:
            - Id: TestLambda1
              FunctionArn: !Ref LambdaVersionArn
              FunctionConfiguration:
                Pinned: 'true'
                Executable: run.exe
                ExecArgs: argument1
                MemorySize: '512'
                Timeout: '2000'
                EncodingType: binary
                Environment:
                  Variables:
                    variable1: value1
                  ResourceAccessPolicies:
                    - ResourceId: ResourceId1
                      Permission: ro
                    - ResourceId: ResourceId2
                      Permission: rw
                  AccessSysfs: 'false'
                  Execution:
                    IsolationMode: GreengrassContainer
                    RunAs:
                      Uid: '1'
                      Gid: '10'
    TestLoggerDefinition:
      Type: 'AWS::Greengrass::LoggerDefinition'
      Properties:
        Name: DemoTestLoggerDefinition
    TestLoggerDefinitionVersion:
      Type: 'AWS::Greengrass::LoggerDefinitionVersion'
      Properties:
        LoggerDefinitionId: !Ref TestLoggerDefinition
        Loggers:
          - Id: TestLogger1
            Type: AWSCloudWatch
            Component: GreengrassSystem
            Level: INFO
TestResourceDefinition:
  Type: 'AWS::Greengrass::ResourceDefinition'
  Properties:
    Name: DemoTestResourceDefinition

TestResourceDefinitionVersion:
  Type: 'AWS::Greengrass::ResourceDefinitionVersion'
  Properties:
    ResourceDefinitionId: !Ref TestResourceDefinition
    Resources:
      - Id: ResourceId1
        Name: LocalDeviceResource
        ResourceDataContainer:
          LocalDeviceResourceData:
            SourcePath: /dev/TestSourcePath1
            GroupOwnerSetting:
              AutoAddGroupOwner: 'false'
              GroupOwner: TestOwner
      - Id: ResourceId2
        Name: LocalVolumeResourceData
        ResourceDataContainer:
          LocalVolumeResourceData:
            SourcePath: /dev/TestSourcePath2
            DestinationPath: /volumes/TestDestinationPath2
            GroupOwnerSetting:
              AutoAddGroupOwner: 'false'
              GroupOwner: TestOwner

TestSubscriptionDefinition:
  Type: 'AWS::Greengrass::SubscriptionDefinition'
  Properties:
    Name: DemoTestSubscriptionDefinition

TestSubscriptionDefinitionVersion:
  Type: 'AWS::Greengrass::SubscriptionDefinitionVersion'
  Properties:
    SubscriptionDefinitionId: !Ref TestSubscriptionDefinition
    Subscriptions:
      - Id: TestSubscription1
        Source: !Join
          - - 'arn:aws:iot'
          - !Ref 'AWS::Region'
          - !Ref 'AWS::AccountId'
          - thing/TestDevice1
        Subject: TestSubjectUpdated
        Target: !Ref LambdaVersionArn

TestGroup:
  Type: 'AWS::Greengrass::Group'
  Properties:
    Name: DemoTestGroupNewName
    RoleArn: !Join
      - - 'arn:aws:iam:'
      - !Ref 'AWS::AccountId'
      - role/TestUser
    InitialVersion:
      CoreDefinitionVersionArn: !Ref TestCoreDefinitionVersion
      DeviceDefinitionVersionArn: !Ref TestDeviceDefinitionVersion
      FunctionDefinitionVersionArn: !Ref TestFunctionDefinitionVersion
      LoggerDefinitionVersionArn: !Ref TestLoggerDefinitionVersion
      ResourceDefinitionVersionArn: !Ref TestResourceDefinitionVersion

Outputs:
  CommandToDeployGroup:
    Value: !Join
      - - ' ':
        - groupVersion=$(cut -d'/' -f6 <<<
          - !GetAtt
        - ' ':
          - !Ref TestResourceDefinitionVersionArn

480
Supported AWS Regions

Currently, you can create and manage AWS IoT Greengrass resources only in the following AWS Regions:

- US East (Ohio)
- US East (N. Virginia)
- US West (Oregon)
- Asia Pacific (Mumbai)
- Asia Pacific (Seoul)
- Asia Pacific (Singapore)
- Asia Pacific (Sydney)
- Asia Pacific (Tokyo)
- China (Beijing)
- EU (Frankfurt)
- EU (Ireland)
- EU (London)
- AWS GovCloud (US-West)
Using AWS IoT Device Tester for AWS IoT Greengrass

You can use AWS IoT Device Tester (IDT) for AWS IoT Greengrass 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 your device can send and receive MQTT messages and process them correctly. IDT for AWS IoT Greengrass generates test reports that you can submit to AWS IoT to add your hardware to the AWS Partner Device Catalog. For more information, see AWS Device Qualification Program.

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.

In addition to testing devices, IDT for AWS IoT Greengrass creates resources (for example, AWS IoT things, AWS IoT Greengrass groups, Lambda functions, and so on) to facilitate the qualification process.

To create these resources, IDT for AWS IoT Greengrass uses the AWS credentials configured in the `config.json` to make API calls on your behalf. These resources are provisioned at various times during a test.

When you run IDT for AWS IoT Greengrass on your host computer, 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.

IDT for AWS IoT Greengrass organizes tests using the concepts of a test suite and test groups.

The test suite is the collection of all test groups used to verify a device works with AWS IoT Greengrass.

A test group consists of all the individual tests related to a feature being tested. For more information, see Test Group Descriptions (p. 502).
AWS IoT Device Tester for AWS IoT Greengrass Versions

The latest version of IDT for AWS IoT Greengrass supports the AWS IoT Greengrass versions listed here. If you need to test an earlier version of AWS IoT Greengrass, see Earlier IDT Versions for AWS IoT Greengrass (p. 483).

By downloading the software, you agree to the AWS IoT Device Tester License Agreement. New releases of AWS IoT Greengrass might require you to download a new version of IDT for AWS IoT Greengrass. You receive a notification when you start a test run if IDT for AWS IoT Greengrass is not compatible with the version of AWS IoT Greengrass you are using.

IDT v2.1.0 for AWS IoT Greengrass v1.9.x, v1.8.x, and v1.7.x

- IDT for AWS IoT Greengrass: Linux
- IDT for AWS IoT Greengrass: macOS
- IDT for AWS IoT Greengrass: Windows

Release Notes:

- Added support for AWS IoT Greengrass v1.9.4.
- Added support for Linux-ARMv6l devices.

Earlier IDT Versions for AWS IoT Greengrass

This section contains links for downloading earlier versions of IDT.

IDT v2.0.0 for AWS IoT Greengrass v1.9.3, v1.9.2, v.1.9.1, v1.9.0, v1.8.4, v1.8.3, and v1.8.2

- IDT for AWS IoT Greengrass: Linux
- IDT for AWS IoT Greengrass: macOS
- IDT for AWS IoT Greengrass: Windows

Release Notes:

- Removed dependency on Python for device under test.
- Test suite execution time reduced by more than 50 percent, which makes the qualification process faster.
- Executable size reduced by more than 50 percent, which makes download and installation faster.
- Improved timeout multiplier support for all test cases.
- Enhanced post-diagnostics messages for faster troubleshooting of errors.
- Updated Permissions Policy Template (p. 485) required to run IDT.
- Added support for AWS IoT Greengrass v1.9.3.

IDT v1.3.3 for AWS IoT Greengrass v1.9.2, v1.9.1, v1.9.0, v1.8.3, and v1.8.2

- IDT for AWS IoT Greengrass: Linux
- IDT for AWS IoT Greengrass: macOS
- IDT for AWS IoT Greengrass: Windows
Release Notes:
- Added support for Greengrass v1.9.2 and v1.8.3.
- Added support for Greengrass OpenWrt.
- Added SSH user name and password device sign-in.
- Added native test bug fix for OpenWrt-ARMv7l platform.

IDT v1.2 for AWS IoT Greengrass v1.8.1
- IDT for AWS IoT Greengrass: Linux
- IDT for AWS IoT Greengrass: macOS
- IDT for AWS IoT Greengrass: Windows

Release Notes:
- Added a configurable timeout multiplier to address and troubleshoot timeout issues (for example, low bandwidth connections).

IDT v1.1 for AWS IoT Greengrass v1.8.0
- IDT for AWS IoT Greengrass: Linux
- IDT for AWS IoT Greengrass: macOS
- IDT for AWS IoT Greengrass: Windows

Release Notes:
- Added support for AWS IoT Greengrass Hardware Security Integration (HSI).
- Added support for AWS IoT Greengrass container and no container.
- Added automated AWS IoT Greengrass service role creation.
- Improved test resource cleanup.
- Added test execution summary report.

IDT v1.1 for AWS IoT Greengrass v1.7.1
- IDT for AWS IoT Greengrass: Linux
- IDT for AWS IoT Greengrass: macOS
- IDT for AWS IoT Greengrass: Windows

Release Notes:
- Added support for AWS IoT Greengrass Hardware Security Integration (HSI).
- Added support for AWS IoT Greengrass container and no container.
- Added automated AWS IoT Greengrass service role creation.
- Improved test resource cleanup.
- Added test execution summary report.

IDT 1.0 (Updated 3/11/19) for AWS IoT Greengrass v1.6.1
- IDT for AWS IoT Greengrass: Linux
Prerequisites

This section describes the prerequisites for using IDT for AWS IoT Greengrass.

Download the Latest Version of AWS IoT Device Tester for AWS IoT Greengrass

Download the latest version of IDT from AWS IoT Device Tester for AWS IoT Greengrass Versions (p. 483). Extract the software into a location on your file system where you have read and write permissions.

Windows has a path length limitation of 260 characters. If you are using Windows, extract IDT for AWS IoT Greengrass into a root directory like `C:\` or `D:\` to keep your paths under the 260 character limit.

Create and Configure an AWS Account

Follow these steps to create and configure an AWS account, create an IAM user, and create an IAM policy that grants IDT for AWS IoT Greengrass permission to access resources on your behalf while running tests.

1. Create an AWS account.
2. Create an IAM policy that grants IDT for AWS IoT Greengrass the permissions it needs to run the tests and collect IDT usage data. Use the policy templates described in Permissions Policy Template (p. 485).
3. Create an IAM user in your AWS account and attach the policy you just created.

Permissions Policy Template

The following is a policy template that grants the permissions required for IDT for AWS IoT Greengrass to run tests.

Important

The following policy template grants permission to create roles, create policies, and attach policies to roles. IDT for AWS IoT Greengrass uses these permissions for tests that need to create roles. Although the policy template does not provide administrator privileges to the user, the permissions can be potentially used to gain administrator access to your AWS account.

IDT v2.0.0

```json
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "VisualEditor0",
         "Effect": "Allow",
         "Action": [
            "iam:CreateRole",
```
"iam:DeleteRole",
"iam:AttachRolePolicy",
"iam:PassRole",
"iam:DetachRolePolicy"
],
"Resource": [
"arn:aws:iam::*:role/idt-*",
"arn:aws:iam::*:role/GreengrassServiceRole"
]
},
{
"Sid": "VisualEditor1",
"Effect": "Allow",
"Action": [
"lambda:DeleteFunction",
"lambda:CreateFunction"
],
"Resource": ["arn:aws:lambda::*:function:idt-*"]
},
{
"Sid": "VisualEditor2",
"Effect": "Allow",
"Action": [
"iot:AttachPolicy",
"iot:DetachPolicy",
"iot:DeletePolicy"
],
"Resource": ["arn:aws:iot::*:policy/idt-*",
"arn:aws:iot::*:cert/*"]
},
{
"Sid": "VisualEditor3",
"Effect": "Allow",
"Action": [
"iot:AttachThingPrincipal",
"iot:DeleteThing",
"iot:DetachThingPrincipal",
"iot:CreateThing"
],
"Resource": ["arn:aws:iot::*:thing/idt-*",
"arn:aws:iot::*:cert/*"]
},
{
"Sid": "VisualEditor4",
"Effect": "Allow",
"Action": [
"iot:DeleteCertificate",
"iot:UpdateCertificate"
],
"Resource": ["arn:aws:iot::*:cert/*"]
},
{
"Sid": "VisualEditor5",
"Effect": "Allow",
"Action": [
"iot:CreateJob",
"iot:DescribeJob",
"iot:DescribeJobExecution",
"iot:DeleteJob"
],
"Resource": ["arn:aws:iot::*:job/*",
"arn:aws:iot::*:cert/*"]
}
"arn:aws:iot::*:thing/idt-*"
],
{
"Sid": "VisualEditor6",
"Effect": "Allow",
"Action": [
"iot:CreatePolicy",
"iot:CreateKeysAndCertificate",
"iot:UpdateThingShadow",
"iot:GetThingShadow",
"iot:DescribeEndpoint",
"greengrass:*",
"iam:ListAttachedRolePolicies",
"iot:CreateCertificateFromCsr",
"iot:ListThings"
],
"Resource": "*"
},
{
"Sid": "VisualEditor7",
"Effect": "Allow",
"Action": ["execute-api:Invoke"],
"Resource": "arn:aws:execute-api:*:098862408343:*"
}
]
}

IDT v1.3.3 and earlier

{
"Version": "2012-10-17",
"Statement": [
{
"Sid": "VisualEditor0",
"Effect": "Allow",
"Action": [
"iam:CreateRole",
"iam:DeleteRole",
"iam:AttachRolePolicy",
"iam:PassRole",
"iam:DetachRolePolicy"
],
"Resource": [
"arn:aws:iam::*:role/idt-*",
"arn:aws:iam::*:role/GreengrassServiceRole"
]
},
{
"Sid": "VisualEditor1",
"Effect": "Allow",
"Action": ["lambda:DeleteFunction",
"lambda:CreateFunction"
],
"Resource": [
"arn:aws:lambda::*:*:function:idt-*"
]
},
{
"Sid": "VisualEditor2",
"Effect": "Allow",
"Action": [
"iot:DeleteCertificate",
"iot:AttachPolicy",
"iot:GetCertificateByCsr",
"iot:CreateCertificateFromCsr"
],
"Resource": "*"
}
]
If you are using AWS IoT Greengrass 1.6.1 or earlier, see Using IDT for Greengrass 1.6.1 and Earlier (p. 488).

Using IDT for Greengrass 1.6.1 and Earlier

If you are using a version of IDT for AWS IoT Greengrass earlier than 1.7.0, you must install the AWS Command Line Interface (CLI) and create a AWS IoT Greengrass service role. In versions 1.7.1 or later, IDT creates the service role for you.

Install the AWS Command Line Interface (CLI)

You use the CLI to perform some operations. If you don't have the CLI installed, follow the instructions in Install the AWS CLI.

Configure the CLI for the AWS Region you want to use by running `aws configure` from a command line. For information about the AWS Regions that support IDT for AWS IoT Greengrass, see AWS Regions and Endpoints.

Configure the AWS IoT Greengrass Service Role

To successfully deploy an AWS IoT Greengrass group, AWS IoT Greengrass requires permissions to perform actions on your behalf. If you are using IDT v1.1 for AWS IoT Greengrass or later, IDT creates and associates the service role with your AWS account for you. If you are using IDT v1.0 for AWS IoT Greengrass, follow these instructions to create and associate the service role with your AWS account.
Configure Your Device

To configure your device you must install AWS IoT Greengrass dependencies, configure the AWS IoT Greengrass Core software, configure your host computer to access your device, and configure user permissions on your device.

Verify AWS IoT Greengrass Dependencies on the Device Under Test

Before IDT for AWS IoT Greengrass can test your devices, make sure that you have set up your device as described in Getting Started with AWS IoT Greengrass. For information about supported platforms, see Supported Platforms.

Configure the AWS IoT Greengrass Software

IDT for AWS IoT Greengrass tests your device for compatibility with a specific version of AWS IoT Greengrass. IDT provides two options for testing AWS IoT Greengrass on your devices:

- Download and use a version of the AWS IoT Greengrass Core software (p. 16). IDT installs the software for you.
- Use a version of the AWS IoT Greengrass Core software already installed on your device.

Note

Each version of AWS IoT Greengrass has a corresponding IDT version. You must download the version of IDT that corresponds to the version of AWS IoT Greengrass you are using.
There are two options for installing AWS IoT Greengrass on your device:

- Download the AWS IoT Greengrass Core software and configure IDT for AWS IoT Greengrass to use it.
- Use an existing installation of the AWS IoT Greengrass Core software.

The following sections describe these options. You only need to do one.

**Option 1: Download the AWS IoT Greengrass Core Software and Configure AWS IoT Device Tester to Use It**

You can download the AWS IoT Greengrass Core software from the AWS IoT Greengrass Core Software (p. 16) downloads page.

1. Find the correct architecture and Linux distribution, and then choose **Download**.
2. Copy the tar.gz file to the `<device-tester-extract-location>/products/greengrass/ggc`.

**Note**
Do not change the name of the AWS IoT Greengrass tar.gz file. Do not place multiple files in this directory for the same operating system and architecture. For example, having both `greengrass-linux-armv7l-1.7.1.tar.gz` and `greengrass-linux-armv7l-1.8.1.tar.gz` files in that directory will cause the tests to fail.

**Option 2: Use an Existing Installation of AWS IoT Greengrass with AWS IoT Device Tester**

Configure IDT to test the AWS IoT Greengrass Core software installed on your device by adding the `greengrassLocation` attribute to the `device.json` file in the `<device_tester_extract_location>/configs` folder. For example:

```
"greengrassLocation" : "<path-to-greengrass-on-device>"
```

For more information about the `device.json` file, see Device Configuration (p. 496).

On Linux devices, the default location of the AWS IoT Greengrass Core software is `/greengrass`.

**Note**
Your device should have an installation of the AWS IoT Greengrass Core software that has not been started.
Make sure you have added the `ggc_user` user and `ggc_group` on your device. For more information, see Environment Setup for AWS IoT Greengrass.

**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 Device Configuration (p. 496).

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.
Configure Your Device

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

In addition to AWS credentials, IDT for AWS IoT Greengrass needs information about the devices that tests are run on (for example, 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:

```
[
    {
        "id": "<pool-id>",
        "sku": "<sku>",
        "features": [
            {
                "name": "os",
                "value": "linux | ubuntu | openwrt"
            },
            {
                "name": "arch",
                "value": "x86_64 | armv7l | aarch64"
            }
        ],
        "hsm": {
            "p11Provider": "</path/to/pkcs11ProviderLibrary>",
            "slotLabel": "<slot-label>",
            "slotUserPin": "<pin>",
            "privateKeyLabel": "<key-label>",
            "openSSLEngine": "</path/to/openssl/engine>"
        },
        "kernelConfigLocation": "",
        "greengrassLocation": ""
    }
]```
Specify **password** only if **method** is set to **password**.

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 boards.

**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**

An array that contains the device’s supported features.

- **Required features:** `os`, `arch`.
- **Supported OS/architecture combinations:**
  - Linux, x86_64
  - Linux, ARMv7l
  - Linux, AArch64
  - Ubuntu, x86_64
  - OpenWRT, ARMv7l
  - OpenWRT, AArch64

**hsm (optional)**

Contains configuration information for testing with an AWS IoT Greengrass Hardware Security Module (HSM). Otherwise, the `<hsm>` element should be omitted. For more information, see Hardware Security Integration (p. 448).

- **hsm.p11Provider**
  
  The absolute path to the PKCS#11 implementation's libdl-loadable library.

- **hsm.slotLabel**
  
  The slot label used to identify the hardware module.

- **hsm.slotUserPin**
  
  The user PIN used to authenticate the AWS IoT Greengrass core to the module.

- **hsm.privateKeyLabel**
  
  The label used to identify the key in the hardware module.

- **hsm.openSSLEngine**
  
  The absolute path to the OpenSSL engine's `.so` file that enables PKCS#11 support on OpenSSL. Used by the AWS IoT Greengrass OTA update agent.

**devices.id**

A user-defined unique identifier for the device being tested.
Configure Your Device

connectivity.protocol

The communication protocol used to communicate with this device. Currently, the only supported value is ssh.

connectivity.ip

The IP address of the device being tested.

connectivity.auth.method

The authorization method used to access a device over the given connectivity protocol. Supported values are:
- pki
- password

connectivity.auth.credentials.password

The password used for signing in to the device being tested. Specify this value 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. Specify this value 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.auth.credentials.privKeyPath

The full path to the private key used to sign in to the device being tested.

greengrassLocation

The location of AWS IoT Greengrass Core software on your devices. This value is only used when you use an existing installation of AWS IoT Greengrass. Use this attribute to tell IDT to use the version of the AWS IoT Greengrass Core software installed on your devices.

kernelConfigLocation

(Optional) The path to the kernel configuration file. AWS IoT Device Tester uses this file to check if the devices have the required kernel features enabled. If not specified, IDT uses the following paths to search for the kernel configuration file: /proc/config.gz and /boot/config-<kernel-version>. AWS IoT Device Tester uses the first path it finds.

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 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
   ```

Setting Configuration 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 Your AWS Credentials

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 user created in `Create and Configure an AWS Account (p. 485)`. You can specify your credentials in one of two ways:

- Credentials file
- 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 to use AWS credentials from your credentials file, edit your `config.json` file as follows:

```
{
    "awsRegion": "us-west-2",
    "auth": {
    "method": "file",
    "credentials": {
        "profile": "default"
    }
```
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 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`:

```bash
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`:

```bash
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:

```json
{
    "awsRegion": "us-west-2",
    "auth": {
        "method": "environment"
    }
}
```

Device Configuration

In addition to AWS credentials, IDT for AWS IoT Greengrass needs information about the devices that tests are run on (for example, 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`:

```json
[
    {
        "id": "<pool-id>",
        "sku": "<sku>",
        "features": [
            {
                "name": "os",
                "value": "linux | ubuntu | openwrt"
            },
            {
                "name": "arch",
                "value": "x86_64 | armv7l | aarch64"
            }
        ]
    }
]
Note
Specify privKeyPath only if method is set to pki.
Specify password only if method is set to password

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 boards.

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

An array that contains the device's supported features.
- Required features: os, arch.
- Supported OS/architecture combinations:
  - Linux, x86_64
  - Linux, ARMv7l
  - Linux, AArch64
  - Ubuntu, x86_64
  - OpenWRT, ARMv7l
• OpenWRT, AArch64

**hsm (optional)**

Contains configuration information for testing with an AWS IoT Greengrass Hardware Security Module (HSM). Otherwise, the `<hsm>` element should be omitted. For more information, see [Hardware Security Integration (p. 448)].

**hsm.p11Provider**

The absolute path to the PKCS#11 implementation's libdl-loadable library.

**hsm.slotLabel**

The slot label used to identify the hardware module.

**hsm.slotUserPin**

The user PIN used to authenticate the AWS IoT Greengrass core to the module.

**hsm,privateKeyLabel**

The label used to identify the key in the hardware module.

**hsm.openSSLEngine**

The absolute path to the OpenSSL engine's `.so` file that enables PKCS#11 support on OpenSSL. Used by the AWS IoT Greengrass OTA update agent.

**devices.id**

A user-defined unique identifier for the device being tested.

**connectivity.protocol**

The communication protocol used to communicate with this device. Currently, the only supported value is `ssh`.

**connectivity.ip**

The IP address of the device being tested.

**connectivity.auth.method**

The authorization method used to access a device over the given connectivity protocol. Supported values are:

• `pki`
• `password`

**connectivity.auth.credentials.password**

The password used for signing in to the device being tested. Specify this value 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. Specify this value 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.auth.credentials.privKeyPath**

The full path to the private key used to sign in to the device being tested.
greengrassLocation

The location of AWS IoT Greengrass Core software on your devices. This value is only used when you use an existing installation of AWS IoT Greengrass. Use this attribute to tell IDT to use the version of the AWS IoT Greengrass Core software installed on your devices.

kernelConfigLocation

(Optional) The path to the kernel configuration file. AWS IoT Device Tester uses this file to check if the devices have the required kernel features enabled. If not specified, IDT uses the following paths to search for the kernel configuration file: /proc/config.gz and /boot/config-<kernel-version>. AWS IoT Device Tester uses the first path it finds.

Running Tests

After you set the required configuration, you can start the tests. The run time 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.

The following example command line shows you how to run the qualification tests for a device pool (a set of identical devices). You can find these commands in the <devicetester-extract-location>/bin directory.

Use the following command to run all test groups in a specified suite:

```
devicetester_[linux | mac | win_x86-64] run-suite --suite-id GGQ_1 --pool-id <pool-id>
```

Use the following command to run a specific test group:

```
devicetester_[linux | mac | win_x86-64] run-suite --suite-id GGQ_1 --group-id <group-id> --pool-id <pool-id>
```

suite-id and pool-id are optional if you are running a single test suite on a single device pool (that is, you have only one device pool defined in your device.json file).

AWS IoT Device Tester for AWS IoT Greengrass Commands

The IDT commands can be used for the following operations:

help

Lists information about the specified command.

list-groups

Lists the groups in a given test suite.

list-suites

Lists the available test suites.

run-suite

Runs a suite of tests on a pool of devices.

Understanding Results and Logs

This section describes how to view and interpret IDT result reports and logs.
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 can be found in `<devicetester-extract-location> /results/<execution-id>/`. Both reports capture the results from the qualification test suite execution.

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 (for example, local resource access, shadow, MQTT, and so on).

The `GGQ_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 execution. 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, it took to run the qualification 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.
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.

```xml
<feature name="aws-iot-greengrass-no-container" value="supported" type="required"/>
```

Features marked as `optional` are not required for qualification. The following snippets show optional features.

```xml
<feature name="aws-iot-greengrass-container" value="supported" type="optional"/>
<feature name="aws-iot-greengrass-hsi" value="not-supported" type="optional"/>
```

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:

```xml
<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:

```xml
<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 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:

```xml
<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 execution 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, `ota.log`)
  - Logs of the test group, including logs from the device under test. When a test fails, a tar.gz file that contains the logs of the device under test for the test is created (for example, `ota_prod_test_1_ggc_logs.tar.gz`).

For more information, see IDT for AWS IoT Greengrass Troubleshooting (p. 504).

Test Group Descriptions

**IDT v2.0.0**

**Container Dependencies**

This test group checks if the device meets all the software and hardware requirements to run Lambda functions in container mode on a AWS IoT Greengrass core.

**Deployment**

This test group validates Lambda functions can be deployed on your device.

**Deployment Container**

This test group validates that Lambda functions can be deployed on the device and run in container mode on a AWS IoT Greengrass core.

**AWS IoT Greengrass core Dependencies**

This test group checks if your device meets all software and hardware requirements for the AWS IoT Greengrass core software.

**Hardware Security Integration (HSI)**

This test group verifies that the provided HSI shared library can interface with the hardware security module (HSM) and implements the required PKCS#11 APIs correctly. The HSM and
shared library must be able to sign a CSR, perform TLS operations, and provide the correct key lengths and public key algorithm.

MQTT

This test group is used to verify the AWS IoT Greengrass message router functionality by checking local communication between AWS IoT Greengrass core and AWS IoT devices.

Over-the-Air (OTA)

This test group validates that your device can successfully perform a AWS IoT Greengrass core OTA update.

Version

This test group checks that the version of AWS IoT Greengrass provided is compatible with the AWS IoT Device Tester version you are using.

IDT v1.3.3 and earlier

Combination (Device Security Interaction)

This test group verifies the functionality of the device certificate manager and IP detection on the AWS IoT Greengrass core device by changing connectivity information on the AWS IoT Greengrass group in the cloud. The test group rotates the AWS IoT Greengrass server certificate and verifies that AWS IoT Greengrass allows connections.

Container Dependencies

The container dependencies test group checks if the device meets all the required dependencies to run Lambda functions in container mode.

Deployment

This test group validates Lambda functions can be deployed on your device.

Device Certificate Manager (DCM)

This test group is used to verify that the AWS IoT Greengrass device certificate manager can generate a server certificate on startup and rotate certificates if they are close to expiration.

AWS IoT Greengrass core Dependencies

This test group checks if your device meets all software and hardware requirements for the AWS IoT Greengrass Core software.

Hardware Security Integration (HSI)

This test group verifies that the provided HSI shared library can interface with the hardware security module (HSM) and implements the required PKCS#11 APIs correctly. The HSM and shared library must be able to sign a CSR, perform TLS operations, and provide the correct key lengths and public key algorithm.

IP Detection (IPD)

This test group is used to verify that core connectivity information is updated when there are IP address changes in an AWS IoT Greengrass core device. For more information, see Activate Automatic IP Detection (p. 59).

Local Resource Access

This test group is used to verify the local resource access feature of AWS IoT Greengrass by providing access to local files and directories owned by various Linux users and groups to containerized Lambda functions through AWS IoT Greengrass LRA APIs. Lambda functions
should be allowed or denied access to local resources based on local resource access configuration.

Logging

This test group is used to verify that the AWS IoT Greengrass logging service can write to a log file using a user Lambda function written in Python.

MQTT

This test group is used to verify the AWS IoT Greengrass message router functionality by sending messages on a topic that is routed to two Lambda functions.

Native

This test group is used to verify that AWS IoT Greengrass can run native (compiled) Lambda functions.

Network

This test group is used to verify that socket connections can be established from a Lambda function. These socket connections should be allowed or denied based on AWS IoT Greengrass core configuration.

Over-the-Air (OTA)

This test group validates that your device can successfully perform an AWS IoT Greengrass core OTA update.

Penetration

This test group checks if the AWS IoT Greengrass Core software fails to start if hard link/soft link protection and seccomp are not enabled. It is also used to verify other security-related features.

Shadow

This test group is used to verify local shadow and shadow cloud-syncing functionality.

Spooler

This test group is used to validate that the MQTT messages are queued with the default spooler configuration.

Token Exchange Service (TES)

This test group is used to verify that AWS IoT Greengrass can exchange its core certificate for valid AWS credentials.

Version

This test group checks that the version of AWS IoT Greengrass provided is compatible with the AWS IoT Device Tester version you are using.

IDT for AWS IoT Greengrass Troubleshooting

IDT for AWS IoT Greengrass writes these errors to various locations based on the type of errors. Errors are written to the console, log files, and test reports.

Error Codes

The following table lists the error codes generated by IDT for AWS IoT Greengrass.
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Code Name</th>
<th>Possible Root Cause</th>
<th>Troubleshooting</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>InternalError</td>
<td>An internal error occurred.</td>
<td>Check logs under the <code>&lt;device-tester-extract-location&gt;/results</code> directory. If you cannot debug the issue, contact AWS Developer Support.</td>
</tr>
<tr>
<td>102</td>
<td>TimeoutError</td>
<td>The test cannot be completed in a limited time range. This can happen if:</td>
<td>• Check the network connection and speed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- There is a slow network connection between the test machine and device</td>
<td>• Make sure that you did not modify any file under the /test directory.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(for example, if you are using a VPN network).</td>
<td>• Try running the failed test group manually with &quot;--group-id&quot; flag.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- A slow network delays the communication between the device and cloud.</td>
<td>• Try running the test suite by increasing the test timeouts. For more information, see Timeout Errors (p. 515).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- The <code>timeout</code> field in test configuration files (<code>test.json</code>) has been mistakenly modified.</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td>PlatformNotSupportError</td>
<td>Incorrect OS/architecture combination specified in <code>device.json</code>.</td>
<td>Change your configuration to one of the supported combinations:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Linux, x86_64</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Linux, ARMv7l</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Linux, AArch64</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Ubuntu, x86_64</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• OpenWRT, ARMv7l</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• OpenWRT, AArch64</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>For more information, see Device Configuration (p. 496).</td>
</tr>
<tr>
<td>104</td>
<td>VersionNotSupportError</td>
<td>The AWS IoT Greengrass Core software version is not supported by the version of IDT you are using.</td>
<td>Use the <code>device_tester_bin version</code> command to find the supported version of the AWS IoT Greengrass</td>
</tr>
<tr>
<td>Error Code</td>
<td>Error Code Name</td>
<td>Possible Root Cause</td>
<td>Troubleshooting</td>
</tr>
<tr>
<td>------------</td>
<td>----------------</td>
<td>---------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Core software. For example, if you are using macOS, use ./devicetester_mac_x86_64 version.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>To find the version of AWS IoT Greengrass Core software you are using:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• If you are running tests with preinstalled AWS IoT Greengrass Core software, use SSH to connect to your AWS IoT Greengrass core device and run &lt;path-to-preinstalled-greengrass-location&gt;/greengrass/ggc/core/greengrassd --version</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• If you are running tests with a different version of the AWS IoT Greengrass Core software, go to the devicetester_greengrass_&lt;os&gt;products/greengrass/gcc directory. The AWS IoT Greengrass Core software version is part of the .zip file name.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>You can test a different version of the AWS IoT Greengrass Core software. For more information, see Getting Started with AWS IoT Greengrass (p. 64).</td>
</tr>
<tr>
<td>Error Code</td>
<td>Error Code Name</td>
<td>Possible Root Cause</td>
<td>Troubleshooting</td>
</tr>
<tr>
<td>------------</td>
<td>------------------------------</td>
<td>-------------------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>105</td>
<td>LanguageNotSupportError</td>
<td>IDT supports Python for AWS IoT Greengrass libraries and SDKs only.</td>
<td>Make sure:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The SDK package under devicetester_greengrass_&lt;os&gt;/products/greengrass/ggsdk is the Python SDK.</td>
<td>• The contents of the bin folder under devicetester_greengrass_&lt;os&gt;/tests/GGQ_1/suite/resources/run.runtimefarm/bin have not been changed.</td>
</tr>
<tr>
<td>106</td>
<td>ValidationError</td>
<td>Some fields in device.json or config.json are invalid.</td>
<td>Check the error message on the right side of the error code in the report.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Invalid private key path: Specify the correct path to your private key. For more information, see Device Configuration (p. 496).</td>
<td>• Invalid private key path: Specify the correct path to your private key. For more information, see Device Configuration (p. 496).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Invalid AWS Region: Specify a valid AWS Region in your config.json file. For more information, see AWS Service Endpoints.</td>
<td>• Invalid AWS Region: Specify a valid AWS Region in your config.json file. For more information, see AWS Service Endpoints.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• AWS credentials: Set valid AWS credentials on your test machine (through environment variables or the credentials file). Verify that the auth field is configured correctly. For more information, see Create and Configure an AWS Account (p. 485).</td>
<td>• AWS credentials: Set valid AWS credentials on your test machine (through environment variables or the credentials file). Verify that the auth field is configured correctly. For more information, see Create and Configure an AWS Account (p. 485).</td>
</tr>
<tr>
<td>Error Code</td>
<td>Error Code Name</td>
<td>Possible Root Cause</td>
<td>Troubleshooting</td>
</tr>
<tr>
<td>------------</td>
<td>----------------------------</td>
<td>-------------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| 107        | SSHConnectionFailed        | The test machine cannot connect to the configured device.                           | Verify the following fields in your device.json file are correct:  
  • ip  
  • user  
  • privKeyPath  
  • password  
  For more information, see Device Configuration (p. 496). |
| 108        | RunCommandError            | A test failed to execute a command on the device under test.                         | Verify that root access is allowed for the configured user in device.json.  
  A password is required by some devices when executing commands with root access. Make sure root access is allowed without a password. For more information, consult the documentation for your device.  
  Try running the failing command manually on your device to see if an error occurs. |
<p>| 109        | PermissionDeniedError      | No root access.                                                                     | Set root access for the configured user on your device. |
| 110        | CreateFileError            | Unable to create a file.                                                             | Check your device's disk space and directory permissions. |
| 111        | CreateDirError             | Unable to create a directory.                                                        | Check your device's disk space and directory permissions. |
| 112        | InvalidPathError           | The path to the AWS IoT Greengrass Core software is incorrect.                      | Verify that the path in the error message is valid. Do not edit any files under the devicetester_greengrass_&lt;os&gt; directory. |</p>
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Code Name</th>
<th>Possible Root Cause</th>
<th>Troubleshooting</th>
</tr>
</thead>
<tbody>
<tr>
<td>113</td>
<td>InvalidFileError</td>
<td>A file is invalid.</td>
<td>Verify that the file in the error message is valid.</td>
</tr>
</tbody>
</table>
| 114        | ReadFileError         | The specified file cannot be read. | Verify the following:  
• File permissions are correct.  
• limits.config allows enough files to be open.  
• The file specified in the error message exists and is valid.  
If you are testing on macOS, increase the open files limit. The default limit is 256, which is enough for testing. |
| 115        | FileNotFoundError    | A required file was not found. | Verify the following:  
• The SDK package exists under devicetester_greengrass_<os>/products/greengrass/ggsdk.  
• The files under devicetester_greengrass_<os>/tests have not been modified. |
| 116        | OpenFileFailed        | Unable to open the specified file. | Verify the following:  
• The file specified in the error message exists and is valid.  
• limits.config allows enough files to be open.  
If you are testing on macOS, increase the open files limit. The default limit is 256, which is enough for testing. |
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Code Name</th>
<th>Possible Root Cause</th>
<th>Troubleshooting</th>
</tr>
</thead>
<tbody>
<tr>
<td>117</td>
<td>WriteFileFailed</td>
<td>Failed to write file (can be the DUT or test machine).</td>
<td>Try to manually create a file in the directory specified in the error message.</td>
</tr>
<tr>
<td>118</td>
<td>FileCleanUpError</td>
<td>A test failed to remove the specified file or directory or to umount the specified file on the remote device.</td>
<td>If the binary file is still running, the file might be locked. End the process and delete the specified file.</td>
</tr>
<tr>
<td>119</td>
<td>InvalidInputError</td>
<td>Invalid configuration.</td>
<td>Verify your suite.json file is valid.</td>
</tr>
<tr>
<td>120</td>
<td>InvalidCredentialError</td>
<td>Invalid AWS credentials.</td>
<td>Verify your AWS credentials. Because this error can also be caused by network problems, check your network connection and rerun the test group.</td>
</tr>
<tr>
<td>121</td>
<td>AWSSessionError</td>
<td>Failed to create an AWS session.</td>
<td>This error can occur if AWS credentials are invalid or the internet connection is unstable. Try using the AWS CLI to call an AWS API operation.</td>
</tr>
<tr>
<td>122</td>
<td>AWSApiCallError</td>
<td>An AWS API error occurred.</td>
<td>This error might be due to a network issue. Check your network before retrying the test group.</td>
</tr>
<tr>
<td>Error Code</td>
<td>Error Code Name</td>
<td>Possible Root Cause</td>
<td>Troubleshooting</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------------</td>
<td>--------------------------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>123</td>
<td>IpNotExistError</td>
<td>IP address is not included in connectivity information.</td>
<td>Check your internet connection. You can use the AWS IoT Greengrass console to check the connectivity information for the AWS IoT Greengrass core thing that is being used by the test. If there are 10 endpoints included in the connectivity information, you can remove some or all of them and rerun the test. For more information, see Connectivity Information.</td>
</tr>
<tr>
<td>124</td>
<td>OTAJobNotCompleteError</td>
<td>An OTA job did not complete.</td>
<td>Check your internet connection and retry the OTA test group.</td>
</tr>
</tbody>
</table>
| 125        | CreateGreengrassServiceRoleError | the following occurred:  
  • An error occurred while creating a role.  
  • An error occurred while attaching a policy to the AWS IoT Greengrass service role.  
  • The policy associated with the service role is invalid.  
  • An error occurred when associating a role with an AWS account. | Configure the AWS IoT Greengrass service role. For more information, see Configure the AWS IoT Greengrass Service Role (p. 488).                                                                                     |
<p>| 126        | DependenciesNotPresentError | One or more dependencies required for the specific test are not present on the device. | Check the test log <code>&lt;device-tester-extract-location&gt;/results/&lt;execution-id&gt;/logs/&lt;test-case-name&gt;.log&gt;</code> to see which dependencies are missing on your device.                                                      |</p>
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Code Name</th>
<th>Possible Root Cause</th>
<th>Troubleshooting</th>
</tr>
</thead>
<tbody>
<tr>
<td>127</td>
<td>InvalidHSMConfiguration</td>
<td>The provided HSM/PKCS configuration is incorrect.</td>
<td>In your device.json file, provide the configuration required to interact with the HSM using PKCS#11.</td>
</tr>
<tr>
<td>128</td>
<td>OTAJobNotSucceededError</td>
<td>The OTA job did not succeed.</td>
<td>Check your internet connectivity and retry the OTA test group. If the problem persists, contact AWS Developer Support.</td>
</tr>
<tr>
<td>129</td>
<td>NoConnectivityError</td>
<td>The host agent is failing to connect to internet.</td>
<td>Check your network connection and firewall settings. Retry the test group after the connectivity issue is resolved.</td>
</tr>
<tr>
<td>130</td>
<td>NoPermissionError</td>
<td>The IAM user you are using to run IDT for AWS IoT Greengrass does not have permission to create the AWS resources required to run IDT.</td>
<td>See Permissions Policy Template for the policy template that grants the permissions required to run IDT for AWS IoT Greengrass.</td>
</tr>
</tbody>
</table>
Resolving IDT for AWS IoT Greengrass Errors

When you use IDT, you must get the correct configuration files in place before you run IDT for AWS IoT Greengrass. If you are getting 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.

Where Do I Look?

High-level errors are displayed on the console during execution, and a summary of the failed tests with the error is displayed when all tests are complete. `awsiotdevicetester_report.xml` contains a summary of all the errors that caused a test to fail. The log files for each test run are stored in a directory named with an UUID for the test execution that was displayed on the console during the test run.

The test logs directory is located in `<device-tester-extract-location>/results/<uuid_execution_id>/logs/`. This directory contains the following files, which are useful for debugging.

### Error Code Table

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Code Name</th>
<th>Possible Root Cause</th>
<th>Troubleshooting</th>
</tr>
</thead>
</table>
| 131        | LeftoverAgentExistError   | Your device is running AWS IoT Greengrass processes when you attempt to start IDT for AWS IoT Greengrass. | Make sure there is no existing Greengrass daemon running on your device.  
  • You can use this command to stop daemon: `sudo ./<absolute-path-to-greengrass-daemon>/greengrassd stop`.  
  • You can also terminate the Greengrass daemon by PID.  

**Note**  
If you are using an existing installation of AWS IoT Greengrass configured to start automatically after reboot, you need to stop the daemon after reboot and before running the test suite.
<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_manager.log</td>
<td>All of the logs that were written to the console during the test execution. A summary of the results is located at the end of this file, which includes a list of which tests failed. The warning and error logs in this file can give you some information about the failures.</td>
</tr>
<tr>
<td>&lt;test-name&gt;.log</td>
<td>Detailed logs for the specific test.</td>
</tr>
<tr>
<td>&lt;test-name&gt;_ggc_logs.tar.gz</td>
<td>A compressed collection of all the logs the AWS IoT Greengrass core daemon generated during the test. For more information, see Troubleshooting AWS IoT Greengrass.</td>
</tr>
<tr>
<td>&lt;test-name&gt;_ota_logs.tar.gz</td>
<td>A compressed collection of logs generated by the AWS IoT Greengrass OTA agent during the test. For OTA tests only.</td>
</tr>
<tr>
<td>&lt;test-name&gt;basic_assertion_publisher_ggad_logs.tar.gz</td>
<td>A compressed collection of logs generated by the AWS IoT publisher device during the test. Generated by the IDT v2.0.0 MQTT tests.</td>
</tr>
<tr>
<td>&lt;test-name&gt;basic_assertion_subscriber_ggad_logs.tar.gz</td>
<td>A compressed collection of logs generated by the AWS IoT subscriber device during the test. Generated by the IDT v2.0.0 MQTT tests.</td>
</tr>
</tbody>
</table>

### Parsing Errors

Occasionally, a typo 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 cannot 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.

### Required Parameter Missing Error

Because new features are being added to IDT, changes to the configuration files might be introduced. Using an old configuration file might break your configuration. If this happens, the <test_case_id>.log file under /results/<uuid>/logs explicitly lists all missing parameters. IDT also validates your JSON configuration file schemas to ensure that the latest supported version has been used.

### Could Not Start Test Error

You might encounter errors that point to failures during test start. There are several possible causes, so do the following:

- Make sure that the pool name you included in your execution command actually exists. The pool name is referenced directly from your device.json file.
- Make sure that the devices in your pool have correct configuration parameters.
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`.

SSH Connection Errors

When IDT cannot connect to a device under test, connection failures are logged in `/results/<uuid>/logs/<test-case-id>.log`. SSH failure 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 setups use the PuTTy terminal application to connect to Linux hosts. This application requires that standard PEM private key files are converted into a proprietary Windows format called PPK. When IDT is configured in your `device.json` file, use PEM files only. If you use a PPK file, IDT cannot create an SSH connection with the AWS IoT Greengrass device and cannot run tests.

Timeout Errors

You can increase the timeout for each test by specifying a timeout multiplier, which is 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 GGQ_1 --pool-id DevicePool1 --timeout-multiplier 2.5
```

For more information, run `run-suite --help`.

Command Not Found Errors While Testing OTA

You need an older version of the OpenSSL library (libssl1.0.0) to run OTA tests on AWS IoT Greengrass devices. Most current Linux distributions use libssl version 1.0.2 or later (v1.1.0). These versions are not compatible with OTA.

For example, on a Raspberry Pi, run the following commands to install the required version of libssl:

1. `wget http://ftp.us.debian.org/debian/pool/main/o/openssl/libssl1.0.0_1.0.2l-1~bpo8+1_armhf.deb`
2. `sudo dpkg -i libssl1.0.0_1.0.2l-1~bpo8+1_armhf.deb`
Troubleshooting AWS IoT Greengrass

This section provides troubleshooting information and possible solutions to help resolve issues with AWS IoT Greengrass.

For information about AWS IoT Greengrass limits, see AWS IoT Greengrass Limits in the Amazon Web Services General Reference.

AWS IoT Greengrass Core Issues

If the AWS IoT Greengrass Core software does not start, try the following general troubleshooting steps:

- Make sure that you install the binaries that are appropriate for your architecture. For more information, see AWS IoT Greengrass Core Software (p. 16).
- Make sure that your core device has local storage available. For more information, see the section called “Troubleshooting Storage Issues” (p. 534).
- Check runtime.log and crash.log for error messages. For more information, see the section called “Troubleshooting with Logs” (p. 534).

Search the following symptoms and errors to find information to help troubleshoot issues with an AWS IoT Greengrass core.

Topics

- Error: The configuration file is missing the CaPath, CertPath or KeyPath. The Greengrass daemon process with [pid = <pid>] died. (p. 517)
- Error: Failed to parse /<greengrass-root>/config/config.json. (p. 518)
- Error: Error occurred while generating TLS config: ErrUnknownURIScheme (p. 518)
- Error: Runtime failed to start: unable to start workers: container test timed out. (p. 518)
- The AWS IoT Greengrass Core software doesn't start after you changed from running with no containerization to running in a Greengrass container. (p. 519)
- Error: Spool size should be at least 262144 bytes. (p. 520)
- Error: Greengrass daemon running with PID: <process-id>. Some system components failed to start. Check 'runtime.log' for errors. (p. 520)
- Device shadow does not sync with the cloud. (p. 520)
Error: The configuration file is missing the CaPath, CertPath or KeyPath. The Greengrass daemon process with [pid = <pid>] died.

Solution: You might see this error in crash.log when the AWS IoT Greengrass Core software does not start. This can occur if you’re running v1.6 or earlier. Do one of the following:

- Upgrade to v1.7 or later. We recommend that you always run the latest version of the AWS IoT Greengrass Core software. For download information, see AWS IoT Greengrass Core Software (p. 16).
- Use the correct config.json format for your AWS IoT Greengrass Core software version. For more information, see the section called “AWS IoT Greengrass Core Configuration File” (p. 24).

Note
To find which version of the AWS IoT Greengrass Core software is installed on the core device, run the following commands in your device terminal.

cd /greengrass-root/ggc/core/
sudo ./greengrassd --version
Error: Failed to parse /<greengrass-root>/config/config.json.

Solution: You might see this error when the AWS IoT Greengrass Core software does not start. Make sure the Greengrass configuration file (p. 24) is using valid JSON format.

Open config.json (located in /<greengrass-root>/config) and validate the JSON format. For example, make sure that commas are used correctly.

Error: Error occurred while generating TLS config: ErrUnknownURIScheme

Solution: You might see this error when the AWS IoT Greengrass Core software does not start. Make sure the properties in the crypto (p. 27) section of the Greengrass configuration file are valid. The error message should provide more information.

Open config.json (located in /<greengrass-root>/config) and check the crypto section. For example, certificate and key paths must use the correct URI format and point to the correct location.

Error: Runtime failed to start: unable to start workers: container test timed out.

Solution: You might see this error when the AWS IoT Greengrass Core software does not start. Set the postStartHealthCheckTimeout property in the Greengrass configuration file (p. 24). This optional property configures the amount of time (in milliseconds) that the Greengrass daemon waits for the post-start health check to finish. The default value is 30 seconds (30000 ms).

Open config.json (located in /<greengrass-root>/config). In the runtime object, add the postStartHealthCheckTimeout property and set the value to a number greater than 30000. Add a comma where needed to create a valid JSON document. For example:

```json
...
"runtime" : {
  "cgroup" : {
    "useSystemd" : "yes"
  },
  "postStartHealthCheckTimeout" : 40000
},
...
```

Solution: You might see this error when the AWS IoT Greengrass Core software does not start. This can occur if you're running AWS IoT Greengrass on a Raspberry Pi and the required memory setup has not been completed. For more information, see this step (p. 69).


Solution: You might see this error when the AWS IoT Greengrass Core software does not start. If you deployed a Lambda executable (p. 187) to the core, check the function's handler property in the group.json file (located in /<greengrass-root>/ggc/deployment/group). If the handler is not the exact name of your compiled executable, replace the contents of the group.json file with an empty JSON object ({}), and run the following commands to start AWS IoT Greengrass:

```
cd /greengrass/ggc/core/
sudo ./greengrassd start
```

Then, use the AWS Lambda API to update the function configuration's handler parameter, publish a new function version, and update the alias. For more information, see AWS Lambda Function Versioning and Aliases.

Assuming that you added the function to your Greengrass group by alias (recommended), you can now redeploy your group. (If not, you must point to the new function version or alias in your group definition and subscriptions before you deploy the group.)

The AWS IoT Greengrass Core software doesn't start after you changed from running with no containerization to running in a Greengrass container.

Solution: Check that you are not missing any container dependencies.
Error: Spool size should be at least 262144 bytes.

Solution: You might see this error when the AWS IoT Greengrass Core software does not start. Open the group.json file (located in `/greengrass-root/ggc/deployment/group`), replace the contents of the file with an empty JSON object `{}`, and run the following commands to start AWS IoT Greengrass:

```
cd /greengrass/ggc/core/
sudo ./greengrassd start
```

Then follow the steps in the section called "To Cache Messages in Local Storage" (p. 56) procedure. For the GGCloudSpooler function, make sure to specify a `GG_CONFIG_MAX_SIZE_BYTES` value that's greater than or equal to 262144.

greengrass/ggc/socket/greengrass_ipc.sock\" to rootfs \"/
greengrass/ggc/packages/<version>/rootfs/merged\" at \"/
greengrass_ipc.sock \" caused \"stat /greengrass/ggc/socket/
greengrass_ipc.sock: permission denied\"\"\".

Solution: You might see this error in runtime.log when the AWS IoT Greengrass Core software does not start. This occurs if your `umask` is higher than 0022. To resolve this issue, you must set the `umask` to 0022 or lower. A value of 0022 grants everyone read permission to new files by default.

Error: Greengrass daemon running with PID: <process-id>. Some system components failed to start. Check 'runtime.log' for errors.

Solution: You might see this error when the AWS IoT Greengrass Core software does not start. Check runtime.log and crash.log for specific error information. For more information, see the section called “Troubleshooting with Logs” (p. 534).

Device shadow does not sync with the cloud.

Solution: Make sure that AWS IoT Greengrass has permissions for `iot:UpdateThingShadow` and `iot:GetThingShadow` actions in the Greengrass service role (p. 443). If the service role uses the `AWSGreengrassResourceAccessRolePolicy` managed policy, these permissions are included by default.
The AWS IoT Greengrass Core software does not run on Raspberry Pi because user namespace is not enabled.

**Solution:** This solution applies to Jessie Raspbian distributions only. Do not run this command on other distributions. User namespaces are enabled in Stretch (and later) distributions by default.

On the Jessie device, run `BRANCH=stable rpi-update` to update to the latest stable versions of the firmware and kernel.

**Note**
User namespace support is required to run AWS IoT Greengrass in the default Greengrass container mode. It isn't required to run in No container mode. For more information, see the section called “Considerations When Choosing Lambda Function Containerization” (p. 181).

**ERROR:** unable to accept TCP connection. accept tcp [::]:8000: accept4: too many open files.

**Solution:** You might see this error in the `greengrassd` script output. This can occur if the file descriptor limit for the AWS IoT Greengrass Core software has reached the threshold and must be increased.

Use the following command and then restart the AWS IoT Greengrass Core software.

```
ulimit -n 2048
```

**Note**
In this example, the limit is increased to 2048. Choose a value appropriate for your use case.

**Error:** Runtime execution error: unable to start lambda container. container_linux.go:259: starting container process caused "process_linux.go:345: container init caused "rootfs_linux.go:50: preparing rootfs caused "permission denied"".

**Solution:** Either install AWS IoT Greengrass directly under the root directory, or make sure that the directory where the AWS IoT Greengrass Core software is installed and its parent directories have execute permissions for everyone.

Solution: AWS IoT Greengrass uses a common handler to validate the properties of all security principals. This warning in runtime.log is expected unless you specified a custom private key for the local MQTT server. For more information, see the section called "Security Principals" (p. 439).


Solution: You might see this error when an over-the-air (OTA) update fails. In the signer role policy, add the target AWS Region as a Resource. This signer role is used to presign the S3 URL for the AWS IoT Greengrass software update. For more information, see S3 URL signer role (p. 150).

The AWS IoT Greengrass core is configured to use a network proxy (p. 46) and your Lambda function can't make outgoing connections.

Solution: Depending on your runtime and the executables used by the Lambda function to create connections, you might also receive connection timeout errors. Make sure your Lambda functions use the appropriate proxy configuration to connect through the network proxy. AWS IoT Greengrass passes the proxy configuration to user-defined Lambda functions through the http_proxy, https_proxy, and no_proxy environment variables. They can be accessed as shown in the following Python snippet.

```python
import os
print(os.environ['HTTP_PROXY'])
```

Note
Most common libraries used to make connections (such as boto3 or cURL and python requests packages) use these environment variables by default.

522
The core is in an infinite connect-disconnect loop. The runtime.log file contains a continuous series of connect and disconnect entries.

**Solution:** This can happen when another device is hard-coded to use the core thing name as the client ID for MQTT connections to AWS IoT. Simultaneous connections in the same AWS Region and AWS account must use unique client IDs. By default, the core uses the core thing name as the client ID for these connections.

To resolve this issue, you can change the client ID used by the other device for the connection (recommended) or override the default value for the core.

**To override the default client ID for the core device**

1. Run the following command to stop the AWS IoT Greengrass daemon:

   ```bash
cd /greengrass-root/ggc/core/
sudo ./greengrassd stop
   
   Open `greengrass-root/config/config.json` for editing as the su user.

2. In the `coreThing` object, add the `coreClientId` property, and set the value to your custom client ID. The value must be between 1 and 128 characters. It must be unique in the current AWS Region for the AWS account.

   ```json
   "coreClientId": "MyCustomClientId"
   ```

3. Start the daemon.

   ```bash
cd /greengrass-root/ggc/core/
sudo ./greengrassd start
   ```

---

**Error:** unable to start lambda container. `container_linux.go:259: starting container process caused "process_linux.go:345: container init caused "rootfs_linux.go:62: mounting ""proc\"" to rootfs \"

**Solution:** On some platforms, you might see this error in `runtime.log` when AWS IoT Greengrass tries to mount the `/proc` file system to create a Lambda container. Or, you might see similar errors, such as `operation not permitted` or `EPERM`. These errors can occur even if tests run on the platform by the dependency checker script pass.

Try one of the following possible solutions:

- Enable the `CONFIG_DEVPTS_MULTIPLE_INSTANCES` option in the Linux kernel.
- Set the `/proc` mount options on the host to `rw,relatim only`. 

---

523
AWS IoT Greengrass Developer Guide

Error: [ERROR]-runtime execution error: unable to start lambda container. {"errorString": "failed to initialize container mounts: failed to create overlay fs for container: mounting overlay at /greengrass/ggc/packages/<ggc-version>/rootfs/merged failed: failed to mount with args source="no_source" dest="/greengrass/ggc/packages/<ggc-version>/rootfs/merged" fstype="overlay" flags="0" data="lowerdir=/greengrass/ggc/packages/<ggc-version>/dns:/,upperdir=/greengrass/ggc/packages/<ggc-version>/rootfs/upper,workdir=/greengrass/ggc/packages/<ggc-version>/rootfs/work": too many levels of symbolic links"}

Solution: You might see this error in runtime.log on a Raspberry Pi if you're running AWS IoT Greengrass Core software v1.9.2 or earlier. (Your software version is shown in the error message.) To resolve this issue, update to AWS IoT Greengrass Core software v1.9.3 or later. For information about using over-the-air updates, see OTA Updates of AWS IoT Greengrass Core Software (p. 148).

Error: [DEBUG]-Failed to get routes. Discarding message.

Solution: Check the subscriptions in your group and make sure that the subscription listed in the [DEBUG] message exists.

Deployment Issues

Use the following information to help troubleshoot deployment issues.

Topics
Your current deployment does not work and you want to revert to a previous working deployment. (p. 525)

- You see a 403 Forbidden error on deployment in the logs. (p. 527)
- A ConcurrentDeployment error occurs when you run the create-deployment command for the first time. (p. 527)
- Error: Greengrass is not authorized to assume the Service Role associated with this account, or the error: Failed: TES service role is not associated with this account. (p. 527)
- The deployment doesn’t finish. (p. 527)
- The deployment doesn’t finish, and runtime.log contains multiple “wait 1s for container to stop” entries. (p. 528)
- Error: Deployment <deployment-id> of type NewDeployment for group <group-id> failed error: Greengrass deployment error: unable to execute download step in deployment. error while processing: unable to load the group file downloaded: could not find UID based on user name, userName: ggc_user. (p. 528)
- Error: Error: [WARN]-MQTT[client] dial tcp: lookup <host-prefix>-ats.iot.<region>.amazonaws.com: no such host ... [ERROR]-Greengrass deployment error: failed to report deployment status back to cloud ... net/http: request canceled while waiting for connection (Client.Timeout exceeded while awaiting headers) (p. 529)

Solution: Use the AWS IoT console or AWS IoT Greengrass API to redeploy a previous working deployment. This deploys the corresponding group version to your core device.

To redeploy a deployment (console)

1. On the group configuration page, choose **Deployments**. This page displays the deployment history for the group, including the date and time, group version, and status of each deployment attempt.
2. Find the row that contains the deployment you want to redeploy. In the **Status** column, choose the ellipsis (...), and then choose **Re-deploy**.

<table>
<thead>
<tr>
<th>Deployed</th>
<th>Version</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jul 1, 2019 1:56:49 PM -0700</td>
<td>8dcd1c859-4ac9-4f5d-afb4-22de0866ef62</td>
<td>Successfully completed ***</td>
</tr>
<tr>
<td>Jul 1, 2019 1:41:47 PM -0700</td>
<td>4ed6e5d-3808-446b-940a-b1a788896382</td>
<td>Successfully completed ***</td>
</tr>
<tr>
<td>Jun 18, 2019 8:16:02 AM -0700</td>
<td>1f5f8706-b350e-dc97-8018-c872e17b335b</td>
<td>Failed ***</td>
</tr>
</tbody>
</table>
To redeploy a deployment (CLI)

1. Use `ListDeployments` to find the ID of the deployment you want to redeploy. For example:

   ```bash
   aws greengrass list-deployments --group-id 74d0b623-c2f2-4cad-9acc-ef92f61fcaf7
   ```

   The command returns the list of deployments for the group.

   ```json
   {   
   "Deployments": [   
   {   
   "DeploymentId": "8d179428-f617-4a77-8a0c-3d61fb8446a6",   
   "DeploymentType": "NewDeployment",   
   "GroupArn": "arn:aws:greengrass:us-west-2:123456789012:/greengrass/groups/74d0b623-c2f2-4cad-9acc-ef92f61fcaf7/versions/8dd1d899-4ac9-4f5d-afe4-22de086efc62",   
   "CreatedAt": "2019-07-01T20:56:49.641Z"   
   },   
   {   
   "DeploymentId": "f8e4c455-8ac4-453a-8252-512dc3e9c596",   
   "DeploymentType": "NewDeployment",   
   "GroupArn": "arn:aws:greengrass:us-west-2:123456789012:/greengrass/groups/74d0b623-c2f2-4cad-9acc-ef92f61fcaf7/versions/4ad66e5d-3808-446b-940a-b1a788898382",   
   "CreatedAt": "2019-07-01T20:56:49.641Z"   
   },   
   {   
   "DeploymentId": "e4aca044-bbd8-41b4-b697-930ca7c40f3e",   
   "DeploymentType": "NewDeployment",   
   "GroupArn": "arn:aws:greengrass:us-west-2:123456789012:/greengrass/groups/74d0b623-c2f2-4cad-9acc-ef92f61fcaf7/versions/1f3870b6-850e-4c97-8018-c872e17b235b",   
   "CreatedAt": "2019-07-01T20:56:49.641Z"   
   }   
   ]   
   }
   ```

   Note

   These AWS CLI commands use example values for the group and deployment ID. When you run the commands, make sure to replace the example values.

2. Use `CreateDeployment` to redeploy the target deployment. Set the deployment type to `Redeployment`. For example:

   ```bash
   aws greengrass create-deployment --deployment-type Redeployment \ 
   --group-id 74d0b623-c2f2-4cad-9acc-ef92f61fcaf7 \ 
   --deployment-id f8e4c455-8ac4-453a-8252-512dc3e9c596
   ```

   The command returns the ARN and ID of the new deployment.

   ```json
   {   
   "DeploymentId": "f9ed02b7-c28e-4df6-83b1-e9553ddd0fc2",   
   "DeploymentArn": "arn:aws:greengrass:us-west-2:123456789012:/greengrass/groups/74d0b623-c2f2-4cad-9acc-ef92f61fcaf7/deployments/f9ed02b7-c28e-4df6-83b1-e9553ddd0fc2"   
   }
   ```

3. Use `GetDeploymentStatus` to get the status of the deployment.
You see a 403 Forbidden error on deployment in the logs.

**Solution:** Make sure the policy of the AWS IoT Greengrass core in the cloud includes "greengrass:*" as an allowed action.

A ConcurrentDeployment error occurs when you run the create-deployment command for the first time.

**Solution:** A deployment might be in progress. You can run `get-deployment-status` to see if a deployment was created. If not, try creating the deployment again.

Error: Greengrass is not authorized to assume the Service Role associated with this account, or the error: Failed: TES service role is not associated with this account.

**Solution:** You might see this error when the deployment fails. Use the `get-service-role-for-account` command in the AWS CLI to check that an appropriate service role is associated with your AWS account in the current AWS Region. To associate a Greengrass service role with your AWS account in the target AWS Region, use `associate-service-role-to-account`. For more information, see the section called “Greengrass Service Role” (p. 443).

The deployment doesn't finish.

**Solution:** Do the following:

- Make sure that the AWS IoT Greengrass daemon is running on your core device. Run the following commands in your core device terminal to check whether the daemon is running and start it, if needed.
  1. To check whether the daemon is running:

     ```bash
     ps aux | grep -E 'greengrass.*daemon'
     ```

     If the output contains a root entry for `/greengrass/ggc/packages/1.9.4/bin/daemon`, then the daemon is running.

     The version in the path depends on the AWS IoT Greengrass Core software version that's installed on your core device.

     2. To start the daemon:

     ```bash
     cd /greengrass/ggc/core/
sudo ./greengrassd start
     ```
• Make sure that your core device is connected and the core connection endpoints are configured properly.

The deployment doesn't finish, and runtime.log contains multiple "wait 1s for container to stop" entries.

**Solution:** Run the following commands in your core device terminal to restart the AWS IoT Greengrass daemon.

```
    cd /greengrass/ggc/core/
    sudo ./greengrassd stop
    sudo ./greengrassd start
```

Error: Deployment <deployment-id> of type NewDeployment for group <group-id> failed error: Error while processing. group config is invalid: 112 or [119 0] don't have rw permission on the file: <path>.

**Solution:** Make sure that the owner group of the <path> directory has read and write permissions to the directory.

Error: <list-of-function-arns> are configured to run as root but Greengrass is not configured to run Lambda functions with root permissions.

**Solution:** You might see this error in runtime.log when the deployment fails. Make sure that you have configured AWS IoT Greengrass to allow Lambda functions to run with root permissions. Either change the value of allowFunctionsToRunAsRoot in greengrass_root/config/config.json to yes or change the Lambda function to run as another user/group. For more information, see the section called “Running a Lambda Function as Root” (p. 180).
Error: Deployment <deployment-id> of type NewDeployment for group <group-id> failed error: Greengrass deployment error: unable to execute download step in deployment. error while processing: unable to load the group file downloaded: could not find UID based on user name, userName: ggc_user: user: unknown user ggc_user.

Solution: If the default access identity (p. 183) of the AWS IoT Greengrass group uses the standard system accounts, the ggc_user user and ggc_group group must be present on the device. For instructions that show how to add the user and group, see this step (p. 69). Make sure to enter the names exactly as shown.


Solution: You might see this error when the deployment fails. Retry the deployment.

Error: [WARN]-MQTT[client] dial tcp: lookup <host-prefix>-ats.iot.<region>.amazonaws.com: no such host ... [ERROR]-Greengrass deployment error: failed to report deployment status back to cloud ... net/http: request canceled while waiting for connection (Client.Timeout exceeded while awaiting headers)

Solution: You might see this error if you're using systemd-resolved, which enables the DNSSEC setting by default. As a result, many public domains are not recognized. Attempts to reach the AWS IoT Greengrass endpoint fail to find the host, so your deployments remain in the In Progress state.

You can use the following commands and output to test for this issue. Replace the region placeholder in the endpoints with your AWS Region.

```bash
$ ping greengrass-ats.iot.region.amazonaws.com
```
Create Group/Create Function Issues

Use the following information to help troubleshoot issues with creating an AWS IoT Greengrass group or Greengrass Lambda function.

Topics
- Error: Your 'IsolationMode' configuration for the group is invalid. (p. 530)
- Error: Your 'IsolationMode' configuration for function with arn <function-arn> is invalid. (p. 531)
- Error: MemorySize configuration for function with arn <function-arn> is not allowed in IsolationMode=NoContainer. (p. 531)
- Error: Access Sysfs configuration for function with arn <function-arn> is not allowed in IsolationMode=NoContainer. (p. 531)
- Error: MemorySize configuration for function with arn <function-arn> is required in IsolationMode=GreengrassContainer. (p. 531)
- Error: Function <function-arn> refers to resource of type <resource-type> that is not allowed in IsolationMode=NoContainer. (p. 531)
- Error: Execution configuration for function with arn <function-arn> is not allowed. (p. 532)

Error: Your 'IsolationMode' configuration for the group is invalid.

Solution: This error occurs when the IsolationMode value in the DefaultConfig of function-definition-version is not supported. Supported values are GreengrassContainer and NoContainer.
Error: Your 'IsolationMode' configuration for function with arn <function-arn> is invalid.

**Solution:** This error occurs when the `IsolationMode` value in the `<function-arn>` of the function-definition-version is not supported. Supported values are `GreengrassContainer` and `NoContainer`.

Error: MemorySize configuration for function with arn <function-arn> is not allowed in IsolationMode=NoContainer.

**Solution:** This error occurs when you specify a `MemorySize` value and you choose to run without containerization. Lambda functions that are run without containerization cannot have memory limits. You can either remove the limit or you can change the Lambda function to run in an AWS IoT Greengrass container.

Error: Access Sysfs configuration for function with arn <function-arn> is not allowed in IsolationMode=NoContainer.

**Solution:** This error occurs when you specify `true` for `AccessSysfs` and you choose to run without containerization. Lambda functions run without containerization must have their code updated to access the file system directly and cannot use `AccessSysfs`. You can either specify a value of `false` for `AccessSysfs` or you can change the Lambda function to run in an AWS IoT Greengrass container.

Error: MemorySize configuration for function with arn <function-arn> is required in IsolationMode=GreengrassContainer.

**Solution:** This error occurs because you did not specify a `MemorySize` limit for a Lambda function that you are running in an AWS IoT Greengrass container. Specify a `MemorySize` value to resolve the error.

Error: Function <function-arn> refers to resource of type <resource-type> that is not allowed in IsolationMode=NoContainer.

function without containerization. If you need those resource types, you must run in an AWS IoT Greengrass container. You can also access local devices directly when running without containerization by changing the code in your Lambda function.

Error: Execution configuration for function with arn <function-arn> is not allowed.

Solution: This error occurs when you create a system Lambda function with GGIPDetector or GGCloudSpooler and you specified IsolationMode or RunAs configuration. You must omit the Execution parameters for this system Lambda function.

Discovery Issues

Use the following information to help troubleshoot issues with the AWS IoT Greengrass Discovery service.

Topics

- Error: Device is a member of too many groups, devices may not be in more than 10 groups (p. 532)

Error: Device is a member of too many groups, devices may not be in more than 10 groups

Solution: This is a known limitation. A Greengrass device (p. 8) can be a member of up to 10 groups.

AWS IoT Greengrass Core in Docker Issues

Use the following information to help troubleshoot issues with running an AWS IoT Greengrass core in a Docker container.

Topics

- Error: Unknown options: -no-include-email (p. 198)
- Warning: IPv4 is disabled. Networking will not work. (p. 198)
- Error: A firewall is blocking file Sharing between windows and the containers. (p. 198)
- Error: Cannot create container for the service greengrass: Conflict. The container name "/aws-iot-greengrass" is already in use. (p. 533)
- Error: [FATAL]-Failed to reset thread's mount namespace due to an unexpected error: "operation not permitted". To maintain consistency, GGC will crash and need to be manually restarted. (p. 533)
Error: Unknown options: -no-include-email

Solution: 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`). If you’re using Windows and you installed the CLI using the MSI installer, you must repeat the installation process. For more information, see Installing the AWS Command Line Interface on Microsoft Windows in the AWS Command Line Interface User Guide.

Warning: IPv4 is disabled. Networking will not work.

Solution: You might receive this warning or a similar message when running AWS IoT Greengrass on a Linux computer. Enable IPv4 network forwarding as described in this step (p. 191). AWS IoT Greengrass cloud deployment and MQTT communications don’t work when IPv4 forwarding isn’t enabled. For more information, see Configure namespaced kernel parameters (sysctls) at runtime in the Docker documentation.

Error: A firewall is blocking file Sharing between windows and the containers.

Solution: You might receive this error or a Firewall Detected message when running Docker on a Windows computer. See the Error: A firewall is blocking file sharing between Windows and the containers Docker support issue. 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: Cannot create container for the service greengrass: Conflict. The container name "/aws-iot-greengrass" is already in use.

Solution: This can occur when the container name is used by an older container. To resolve this issue, run the following command to remove the old Docker container:

```bash
docker rm -f $(docker ps -a -q -f "name=aws-iot-greengrass")
```

Error: [FATAL]-Failed to reset thread's mount namespace due to an unexpected error: "operation not permitted". To maintain consistency, GGC will crash and need to be manually restarted.

Solution: This error in runtime.log can occur when you try to deploy a GreengrassContainer Lambda function to an AWS IoT Greengrass core running in a Docker container. Currently, only NoContainer Lambda functions can be deployed to a Greengrass Docker container.
To resolve this issue, make sure that all Lambda functions are in NoContainer mode (p. 182) and start a new deployment. Then, when starting the container, don't bind-mount the existing deployment directory onto the AWS IoT Greengrass core Docker container. Instead, create an empty deployment directory in its place and bind-mount that in the Docker container. This allows the new Docker container to receive the latest deployment with Lambda functions running in NoContainer mode.

For more information, see the section called “Run AWS IoT Greengrass in a Docker Container” (p. 189).

Troubleshooting with Logs

If logs are configured to be stored on the local file system, start looking in the following locations. Reading the logs on the file system requires root permissions.

`greengrass-root/ggc/var/log/crash.log`

Shows messages generated when an AWS IoT Greengrass core crashes.

`greengrass-root/ggc/var/log/system/runtime.log`

Shows messages about which component failed.

`greengrass-root/ggc/var/log/system/`

Contains all logs from AWS IoT Greengrass system components, such as the certificate manager and the connection manager. By using the messages in `ggc/var/log/system/` and `ggc/var/log/system/runtime.log`, you should be able to find out which error occurred in AWS IoT Greengrass system components.

`greengrass-root/ggc/var/log/user/`

Contains all logs from user-defined Lambda functions. Check this folder to find error messages from your local Lambda functions.

**Note**

By default, `greengrass-root` is the `/greengrass` directory. If a write directory (p. 52) is configured, then the logs are under that directory.

If the logs are configured to be stored on the cloud, use CloudWatch Logs to view log messages. `crash.log` is found only in file system logs on the AWS IoT Greengrass core device.

If AWS IoT is configured to write logs to CloudWatch, check those logs if connection errors occur when system components attempt to connect to AWS IoT.

For more information about AWS IoT Greengrass logging, see Monitoring (p. 457).

**Note**

Logs for AWS IoT Greengrass Core software v1.0 are stored under the `greengrass-root/var/log` directory.

Troubleshooting Storage Issues

When the local file storage is full, some components might start failing:

- Local shadow updates do not occur.
- New AWS IoT Greengrass core MQTT server certificates cannot be downloaded locally.
- Deployments fail.
You should always be aware of the amount of free space available locally. You can calculate free space based on the sizes of deployed Lambda functions, the logging configuration (see Troubleshooting with Logs (p. 534)), and the number of shadows stored locally.

**Troubleshooting Messages**

All messages sent in AWS IoT Greengrass are sent with QoS 0. By default, AWS IoT Greengrass stores messages in an in-memory queue. Therefore, unprocessed messages are lost when the AWS IoT Greengrass core restarts (for example, after a group deployment or device reboot). However, you can configure AWS IoT Greengrass (v1.6 or later) to cache messages to the file system so they persist across core restarts. You can also configure the queue size. For more information, see the section called “MQTT Message Queue” (p. 55).

**Note**

When using the default in-memory queue, we recommend that you deploy groups or restart the device when the service disruption is the lowest.

If you configure a queue size, make sure that it’s greater than or equal to 262144 bytes (256 KB). Otherwise, AWS IoT Greengrass might not start properly.

**Troubleshooting Shadow Synchronization Timeout Issues**

Significant delays in communication between a Greengrass core device and the cloud might cause shadow synchronization to fail because of a timeout. In this case, you should see log entries similar to the following:

```
[2017-07-20T10:01:58.006Z][ERROR]-cloud_shadow_client.go:57,Cloud shadow client error: unable to get cloud shadow what_the_thi
[2017-07-20T10:01:58.006Z][WARN]-sync_manager.go:263,Failed to get cloud copy: Get
[2017-07-20T10:01:58.006Z][ERROR]-sync_manager.go:375,Failed to execute sync operation
```

A possible fix is to configure the amount of time that the core device waits for a host response. Open the config.json (p. 24) file in greengrass-root/config and add a system.shadowSyncTimeout field with a timeout value in seconds. For example:

```
{  
  "system": {  
    "shadowSyncTimeout": 10  
  },  
  "coreThing": {  
    "caPath": "root-ca.pem",  
    "certPath": "cloud.pem.crt",  
    "keyPath": "cloud.pem.key",  
    ...  
  },  
  ...  
}
```
If no `shadowSyncTimeout` value is specified in `config.json`, the default is 5 seconds.

**Note**
For AWS IoT Greengrass Core software v1.6 and earlier, the default `shadowSyncTimeout` is 1 second.

---

**Check the AWS IoT Greengrass Forum**

If you're unable to resolve your issue using the troubleshooting information in this topic, you can search the [AWS IoT Greengrass Forum](https://aws.amazon.com/community/aws-iot-greengrass/) for related issues or post a new forum thread. Members of the AWS IoT Greengrass team actively monitor the forum.
## Document History for AWS IoT Greengrass

The following table describes important changes to the AWS IoT Greengrass Developer Guide after June 2018. For notification about updates to this documentation, you can subscribe to an RSS feed.

<table>
<thead>
<tr>
<th>update-history-change</th>
<th>update-history-description</th>
<th>update-history-date</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Console Support for Deployment Notifications</strong></td>
<td>Use the Amazon EventBridge console to create event rules that trigger when your Greengrass group deployments change state.</td>
<td>November 14, 2019</td>
</tr>
<tr>
<td><strong>AWS IoT Greengrass Version 1.9.4 Released</strong></td>
<td>Version 1.9.4 of the AWS IoT Greengrass Core software is available. This version contains performance improvements and bug fixes. As a best practice, we recommend that you always run the latest version.</td>
<td>October 17, 2019</td>
</tr>
<tr>
<td><strong>Console Support for Managing the Greengrass Service Role</strong></td>
<td>Use new and improved features in the AWS IoT console to manage your Greengrass service role.</td>
<td>October 4, 2019</td>
</tr>
<tr>
<td><strong>Console Support for Managing Group-level Tags</strong></td>
<td>You can create, view, and manage tags for your Greengrass groups in the AWS IoT console.</td>
<td>September 23, 2019</td>
</tr>
<tr>
<td><strong>New Machine Learning Connectors</strong></td>
<td>Use the ML Feedback connector to publish model input and predictions and the ML Object Detection connector to run a local object detection inference service.</td>
<td>September 19, 2019</td>
</tr>
<tr>
<td><strong>AWS IoT Greengrass Version 1.9.3 Released</strong></td>
<td>Version 1.9.3 of the AWS IoT Greengrass Core software is available. This version allows you to install the AWS IoT Greengrass Core software on Raspbian distributions on Armv6l architectures, supports OTA updates on port 443 with ALPN, and contains a bug fix for binary payloads sent from Python 2.7 Lambda functions to other Lambda functions.</td>
<td>September 12, 2019</td>
</tr>
<tr>
<td>AWS IoT Greengrass Version</td>
<td>Description</td>
<td>Date</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>1.8.4 Released</td>
<td>Version 1.8.4 of the AWS IoT Greengrass Core software is available. This version contains performance improvements and bug fixes. If you're running v1.8.x, we recommend that you upgrade to v1.8.4 or v1.9.3. For earlier versions, we recommend that you upgrade to v1.9.3.</td>
<td>August 30, 2019</td>
</tr>
<tr>
<td>1.9.2 Released With Support for OpenWrt</td>
<td>Version 1.9.2 of the AWS IoT Greengrass Core software is available. This version allows you to install the AWS IoT Greengrass Core software on OpenWrt distributions with Armv8 (AArch64) and Armv7l architectures.</td>
<td>June 20, 2019</td>
</tr>
<tr>
<td>1.8.3 Released</td>
<td>Version 1.8.3 of the AWS IoT Greengrass Core software is available. This version contains general performance improvements and bug fixes. If you're running v1.8.x, we recommend that you upgrade to v1.8.3 or v1.9.2. For earlier versions, we recommend that you upgrade to v1.9.2.</td>
<td>June 20, 2019</td>
</tr>
<tr>
<td>1.9.1 Released</td>
<td>Version 1.9.1 of the AWS IoT Greengrass Core software is available. This version contains a bug fix for messages from AWS IoT that contain a wildcard character in the topic.</td>
<td>May 10, 2019</td>
</tr>
<tr>
<td>1.8.2 Released</td>
<td>Version 1.8.2 of the AWS IoT Greengrass Core software is available. This version contains general performance improvements and bug fixes. If you're running v1.8.x, we recommend that you upgrade to v1.8.2 or v1.9.0. For earlier versions, we recommend that you upgrade to v1.9.0.</td>
<td>May 2, 2019</td>
</tr>
<tr>
<td>1.9.0 Released</td>
<td>New features: Support for Python 3.7 and Node.js 8.10 Lambda runtimes, optimized MQTT connections, and Elliptic Curve (EC) key support for the local MQTT server.</td>
<td>May 1, 2019</td>
</tr>
<tr>
<td>Issue</td>
<td>Description</td>
<td>Date</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td><strong>AWS IoT Greengrass Version 1.8.1 Released</strong></td>
<td>Version 1.8.1 of the AWS IoT Greengrass Core software is available. This version contains general performance improvements and bug fixes. As a best practice, we recommend that you always run the latest version.</td>
<td>April 18, 2019</td>
</tr>
<tr>
<td><strong>AWS IoT Greengrass Snap Available on Snapcraft</strong></td>
<td>Use the AWS IoT Greengrass Snap Store app to quickly design, test, and deploy software on Linux devices with AWS IoT Greengrass.</td>
<td>April 1, 2019</td>
</tr>
<tr>
<td><strong>Support for More Access Control Using Tag-Based Permissions</strong></td>
<td>You can use tags in AWS Identity and Access Management (IAM) policies to control access to your AWS IoT Greengrass resources.</td>
<td>March 29, 2019</td>
</tr>
<tr>
<td><strong>IoT Analytics Connector Released</strong></td>
<td>Use the IoT Analytics connector to send local device data to AWS IoT Analytics channels.</td>
<td>March 15, 2019</td>
</tr>
<tr>
<td><strong>Batch Support in Kinesis Firehose Connector</strong></td>
<td>The Kinesis Firehose connector supports sending batched data records to Amazon Kinesis Data Firehose at a specified interval.</td>
<td>March 15, 2019</td>
</tr>
<tr>
<td><strong>AWS CloudFormation Support for AWS IoT Greengrass Resources</strong></td>
<td>Use AWS CloudFormation templates to create and manage AWS IoT Greengrass resources.</td>
<td>March 15, 2019</td>
</tr>
<tr>
<td><strong>AWS IoT Greengrass Version 1.8.0 Released</strong></td>
<td>New features: Configurable default access identity for Lambda functions, support for HTTPS traffic over port 443, and predictably named client IDs for MQTT connections with AWS IoT.</td>
<td>March 7, 2019</td>
</tr>
<tr>
<td><strong>AWS IoT Greengrass Versions 1.7.1 and 1.6.1 Released</strong></td>
<td>Versions 1.7.1 and 1.6.1 of the AWS IoT Greengrass Core software are available. These versions require Linux kernel version 3.17 or later. We recommend that customers running any version of the Greengrass core software upgrade to version 1.7.1 immediately.</td>
<td>February 11, 2019</td>
</tr>
<tr>
<td><strong>Amazon SageMaker Neo Deep Learning Runtime</strong></td>
<td>The Amazon SageMaker Neo deep learning runtime supports machine learning models that have been optimized by the Amazon SageMaker Neo deep learning compiler.</td>
<td>November 28, 2018</td>
</tr>
<tr>
<td><strong>AWS IoT Greengrass Developer Guide</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Run AWS IoT Greengrass in a Docker container</strong></td>
<td>You can run AWS IoT Greengrass in a Docker container by configuring your Greengrass group to run with no containerization.</td>
<td></td>
</tr>
<tr>
<td><strong>AWS IoT Greengrass Version 1.7.0 Released</strong></td>
<td>New features: Greengrass connectors, local secrets manager, isolation and permission settings for Lambda functions, hardware root of trust security, connection using ALPN or network proxy, and Raspbian Stretch support.</td>
<td></td>
</tr>
<tr>
<td><strong>AWS IoT Greengrass Software Downloads</strong></td>
<td>The AWS IoT Greengrass Core software, AWS IoT Greengrass Core SDK, and AWS IoT Greengrass Machine Learning SDK packages are available for download through Amazon CloudFront.</td>
<td></td>
</tr>
<tr>
<td><strong>AWS IoT Device Tester for AWS IoT Greengrass</strong></td>
<td>Use AWS IoT Device Tester for AWS IoT Greengrass to verify that your CPU architecture, kernel configuration, and drivers work with AWS IoT Greengrass.</td>
<td></td>
</tr>
<tr>
<td><strong>AWS CloudTrail Logging for AWS IoT Greengrass API Calls</strong></td>
<td>AWS IoT Greengrass 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.</td>
<td></td>
</tr>
<tr>
<td><strong>Support for TensorFlow v1.10.1 on NVIDIA Jetson TX2</strong></td>
<td>The TensorFlow precompiled library for NVIDIA Jetson TX2 that AWS IoT Greengrass provides now uses TensorFlow v1.10.1. This supports Jetpack 3.3 and CUDA Toolkit 9.0.</td>
<td></td>
</tr>
<tr>
<td><strong>Support for MXNet v1.2.1 Machine Learning Resources</strong></td>
<td>AWS IoT Greengrass supports machine learning models that are trained using MXNet v1.2.1.</td>
<td></td>
</tr>
<tr>
<td><strong>AWS IoT Greengrass Version 1.6.0 Released</strong></td>
<td>New features: Lambda executables, configurable message queue, configurable reconnect retry interval, volume resources under /proc, and configurable write directory.</td>
<td></td>
</tr>
</tbody>
</table>
## Earlier Updates

The following table describes important changes to the AWS IoT Greengrass Developer Guide before July 2018.

<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
</table>
| AWS IoT Greengrass Version 1.5.0 Released | New features:  
  - Local machine learning inference using cloud-trained models. For more information, see Perform Machine Learning Inference (p. 221).  
  - Greengrass Lambda functions support binary input data, in addition to JSON. For more information, see AWS IoT Greengrass Core versions (p. 2). | March 29, 2018 |
| AWS IoT Greengrass Version 1.3.0 Released | New features:  
  - Over-the-air (OTA) update agent capable of handling cloud-deployed, Greengrass update jobs. For more information, see OTA Updates of AWS IoT Greengrass Core Software (p. 148).  
  - Access local peripherals and resources from Greengrass Lambda functions. For more information, see Access Local Resources with Lambda Functions and Connectors (p. 200). | November 27, 2017 |
| AWS IoT Greengrass Version 1.1.0 Released | New features:  
  - Reset deployed AWS IoT Greengrass groups. For more information, see Reset Deployments (p. 163).  
  - Support for Node.js 6.10 and Java 8 Lambda runtimes, in addition to Python 2.7. | September 20, 2017 |
| AWS IoT Greengrass Version 1.0.0 Released | AWS IoT Greengrass is generally available. | June 7, 2017 |