AWS Greengrass
Developer Guide
# Table of Contents

What Is AWS Greengrass? .......................................................................................................................... 1
AWS Greengrass Core Software ........................................................................................................... 2
AWS Greengrass Core Versions ........................................................................................................... 2
AWS Greengrass Groups ....................................................................................................................... 4
Device in AWS Greengrass .................................................................................................................... 5
SDKs ..................................................................................................................................................... 6
Supported Platforms and Requirements ............................................................................................. 12
We Want to Hear from You ................................................................................................................. 16
AWS Greengrass Core .......................................................................................................................... 2
AWS Greengrass Core Configuration File ............................................................................................ 17
Write Directory ................................................................................................................................. 23
MQTT Message Queue ....................................................................................................................... 25
Start Greengrass on System Boot ....................................................................................................... 28
Getting Started with AWS Greengrass .............................................................................................. 29
Requirements ........................................................................................................................................ 29
Create an AWS Account ..................................................................................................................... 30
Module 1: Environment Setup for Greengrass ................................................................................. 30
Setting Up a Raspberry Pi ..................................................................................................................... 30
Setting Up an Amazon EC2 Instance ................................................................................................... 35
Setting Up Other Devices .................................................................................................................... 39
Module 2: Installing the Greengrass Core Software ......................................................................... 43
Configure AWS Greengrass on AWS IoT ............................................................................................ 43
Start AWS Greengrass on the Core Device ....................................................................................... 48
Module 3 (Part 1): Lambda Functions on AWS Greengrass .............................................................. 51
Create and Package a Lambda Function ............................................................................................ 51
Configure the Lambda Function for AWS Greengrass ....................................................................... 57
Deploy Cloud Configurations to a Core Device ................................................................................... 62
Verify the Lambda Function Is Running on the Device ....................................................................... 63
Module 3 (Part 2): Lambda Functions on AWS Greengrass .............................................................. 66
Create and Package the Lambda Function ........................................................................................... 66
Configure Long-Lived Lambda Functions for AWS Greengrass ......................................................... 68
Test Long-Lived Lambda Functions .................................................................................................... 69
Test On-Demand Lambda Functions .................................................................................................. 74
Module 4: Interacting with Devices in an AWS Greengrass Group ..................................................... 77
Create AWS IoT Devices in an AWS Greengrass Group .................................................................... 78
Configure Subscriptions ....................................................................................................................... 82
Install the AWS IoT Device SDK for Python ....................................................................................... 83
Test Communications .......................................................................................................................... 87
Module 5: Interacting with Device Shadows ....................................................................................... 91
Configure Devices and Subscriptions ................................................................................................. 92
Download Required Files .................................................................................................................... 94
Test Communications (Device Syncs Disabled) .................................................................................... 95
Test Communications (Device Syncs Enabled) ................................................................................... 96
Module 6: Accessing AWS Cloud Services .......................................................................................... 98
Configure IAM Roles ............................................................................................................................ 99
Create and Configure the Lambda Function ......................................................................................... 103
Configure Subscriptions ....................................................................................................................... 108
Test Communications .......................................................................................................................... 111
OTA Updates of AWS Greengrass Core Software ............................................................................. 114
Greengrass OTA Agent ....................................................................................................................... 115
Integration with Init systems ............................................................................................................. 117
OTA Self-Update with Managed Respawn ......................................................................................... 118
AWS Greengrass Core Update with Managed Respawn ...................................................................... 118
OTA Agent Self-Update ....................................................................................................................... 118
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greengrass Core Software Update</td>
<td>119</td>
</tr>
<tr>
<td>Reset Deployments</td>
<td>120</td>
</tr>
<tr>
<td>Run Local Lambda Functions</td>
<td>122</td>
</tr>
<tr>
<td>SDKs</td>
<td>122</td>
</tr>
<tr>
<td>- Migrating Cloud-Based Lambda Functions</td>
<td>123</td>
</tr>
<tr>
<td>- Function Aliases and Versions</td>
<td>123</td>
</tr>
<tr>
<td>- Group-Specific Configuration</td>
<td>124</td>
</tr>
<tr>
<td>Communication Flows</td>
<td>124</td>
</tr>
<tr>
<td>- MQTT Messages</td>
<td>125</td>
</tr>
<tr>
<td>- Other Communication Flows</td>
<td>125</td>
</tr>
<tr>
<td>Lifecycle Configuration</td>
<td>125</td>
</tr>
<tr>
<td>Lambda Executables</td>
<td>126</td>
</tr>
<tr>
<td>- Create a Lambda Executable</td>
<td>127</td>
</tr>
<tr>
<td>Access Local Resources with Lambda Functions</td>
<td>129</td>
</tr>
<tr>
<td>Supported Resource Types</td>
<td>129</td>
</tr>
<tr>
<td>Requirements</td>
<td>130</td>
</tr>
<tr>
<td>- Volume Resources Under the /proc Directory</td>
<td>130</td>
</tr>
<tr>
<td>Group Owner File Access Permission</td>
<td>131</td>
</tr>
<tr>
<td>Using the CLI</td>
<td>131</td>
</tr>
<tr>
<td>- Create Local Resources</td>
<td>131</td>
</tr>
<tr>
<td>- Create the Greengrass Function</td>
<td>132</td>
</tr>
<tr>
<td>- Add the Lambda Function to the Group</td>
<td>133</td>
</tr>
<tr>
<td>- Troubleshooting</td>
<td>135</td>
</tr>
<tr>
<td>Using the Console</td>
<td>135</td>
</tr>
<tr>
<td>- Prerequisites</td>
<td>136</td>
</tr>
<tr>
<td>- Create a Lambda Function Deployment Package</td>
<td>136</td>
</tr>
<tr>
<td>- Create and Publish a Lambda Function</td>
<td>138</td>
</tr>
<tr>
<td>- Add the Lambda Function to the Group</td>
<td>142</td>
</tr>
<tr>
<td>- Add a Local Resource to the Group</td>
<td>145</td>
</tr>
<tr>
<td>- Add Subscriptions to the Group</td>
<td>147</td>
</tr>
<tr>
<td>- Deploy the Group</td>
<td>149</td>
</tr>
<tr>
<td>- Test Local Resource Access</td>
<td>150</td>
</tr>
<tr>
<td>Perform Machine Learning Inference</td>
<td>154</td>
</tr>
<tr>
<td>How AWS Greengrass ML Inference Works</td>
<td>154</td>
</tr>
<tr>
<td>Machine Learning Resources</td>
<td>155</td>
</tr>
<tr>
<td>- Supported Model Sources</td>
<td>155</td>
</tr>
<tr>
<td>Requirements</td>
<td>156</td>
</tr>
<tr>
<td>Precompiled Libraries for ML Frameworks</td>
<td>157</td>
</tr>
<tr>
<td>- MXNet Versioning</td>
<td>157</td>
</tr>
<tr>
<td>- TensorFlow Model-Serving Limitations on Raspberry Pi</td>
<td>157</td>
</tr>
<tr>
<td>How to Configure Amazon SageMaker to Use MXNet v0.11</td>
<td>158</td>
</tr>
<tr>
<td>How to Configure Machine Learning Inference</td>
<td>159</td>
</tr>
<tr>
<td>- Prerequisites</td>
<td>160</td>
</tr>
<tr>
<td>- Configure the Raspberry Pi</td>
<td>160</td>
</tr>
<tr>
<td>- Install the MXNet Framework</td>
<td>160</td>
</tr>
<tr>
<td>- Create a Model Package</td>
<td>162</td>
</tr>
<tr>
<td>- Create and Publish a Lambda Function</td>
<td>162</td>
</tr>
<tr>
<td>- Add the Lambda Function to the Group</td>
<td>167</td>
</tr>
<tr>
<td>- Add Resources to the Group</td>
<td>169</td>
</tr>
<tr>
<td>- Add a Subscription to the Group</td>
<td>172</td>
</tr>
<tr>
<td>- Deploy the Group</td>
<td>173</td>
</tr>
<tr>
<td>- Test the App</td>
<td>175</td>
</tr>
<tr>
<td>- Next Steps</td>
<td>178</td>
</tr>
<tr>
<td>- Configuring an NVIDIA Jetson TX2</td>
<td>178</td>
</tr>
<tr>
<td>Greengrass Discovery RESTful API</td>
<td>179</td>
</tr>
<tr>
<td>Request</td>
<td>179</td>
</tr>
</tbody>
</table>
What Is AWS Greengrass?

AWS Greengrass is software that extends AWS Cloud capabilities to local devices, making it possible for those devices to collect and analyze data closer to the source of information, while also securely communicating with each other on local networks. More specifically, developers who use AWS Greengrass can author serverless code (AWS Lambda functions) in the cloud and conveniently deploy it to devices for local execution of applications.

The following diagram shows the basic architecture of AWS Greengrass.

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

In AWS Greengrass, devices securely communicate on a local network and exchange messages with each other without having to connect to the cloud. AWS 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 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 Greengrass provides secure, over-the-air software updates of Lambda functions.

AWS Greengrass consists of:

- Software distributions
  - AWS Greengrass core software
  - AWS Greengrass core SDK
AWS Greengrass Core Software

The AWS Greengrass core software provides the following functionality:

- Allows deployment and execution of local applications created using Lambda functions and managed through the deployment API.
- Enables local messaging between devices over a secure network using a managed subscription scheme through the MQTT protocol.
- Ensures secure connections between devices and the cloud using device authentication and authorization.
- Provides secure, over-the-air software updates of user-defined Lambda functions.

The AWS Greengrass core software consists of:

- A message manager that routes messages between devices, Lambda functions, and AWS IoT.
- A Lambda runtime that runs user-defined Lambda functions.
- An implementation of the Device Shadow service that provides a local copy of shadows, which represent your devices. Shadows can be configured to sync with the cloud.
- A deployment agent that is notified of new or updated AWS Greengrass group configuration. When a new or updated configuration is detected, the deployment agent downloads the configuration data and restarts the AWS Greengrass core.

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

AWS Greengrass Core Versions

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

GGC v1.6.0

Current version.

New features:

- Lambda executables that run binary code on the Greengrass core. Use the new AWS Greengrass Core SDK for C to write Lambda executables in C and C++. For more information, see the section called "Lambda Executables" (p. 126).
• 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. 25).

• Configurable maximum reconnect retry interval for when the core device is disconnected. For more information, see the mqttMaxConnectionRetryInterval property in the section called “AWS Greengrass Core Configuration File” (p. 17).

• Local resource access to the host /proc directory. For more information, see Access Local Resources with Lambda Functions (p. 129).

• Configurable write directory. The AWS Greengrass core software can be deployed to read-only and read-write locations. For more information, see the section called “Write Directory” (p. 23).

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.

GGC v1.5.0

New features:

• AWS Greengrass Machine Learning (ML) Inference is generally available. You can perform ML inference locally on AWS Greengrass devices using models that are built and trained in the cloud. For more information, see Perform Machine Learning Inference (p. 154).

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

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.

GGC v1.3.0

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 Greengrass Core Software (p. 114).

• 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 (p. 129).

GGC v1.1.0

New features:
Deployed AWS Greengrass groups can be reset by deleting Lambda functions, subscriptions, and configurations. For more information, see *Reset Deployments* (p. 120).

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 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.

GGC v1.0.0

Initial version.

AWS Greengrass Groups

An AWS Greengrass group definition is a collection of settings for AWS Greengrass core devices and the devices that communicate with them. The following diagram shows the objects that make up an AWS Greengrass group.

In the preceding diagram:

- **A**: AWS Greengrass group definition
  
  A collection of information about the AWS Greengrass group.
B: AWS Greengrass group settings

These include:
• AWS Greengrass group role.
• Log configuration.
• Certification authority and local connection configuration.
• AWS Greengrass core connectivity information.

C: AWS Greengrass core

The AWS IoT thing that represents the AWS Greengrass core.

D: Lambda function definition

A list of Lambda functions to be deployed to the AWS Greengrass core of the group, with associated configuration data. For more information, see Run Local Lambda Functions (p. 122).

E: Subscription definition

A list of subscriptions that enable communication using MQTT messages. A subscription defines:
• A message source. This can be an AWS IoT thing (device), Lambda function, AWS IoT, or the local AWS Greengrass shadow service.
• A subject, which is an MQTT topic or topic filter that's used to filter message data.
• A message target, which identifies the destination for messages published by the message source. This can be an AWS IoT thing (device), Lambda function, AWS IoT, or the local AWS Greengrass shadow service.

For more information, see the section called “Greengrass Messaging Workflow” (p. 188).

F: Device definition

A list containing an AWS Greengrass core and AWS IoT things that are members of the AWS Greengrass group, with associated configuration data. This data specifies which devices are AWS Greengrass cores and which devices should sync shadow data with AWS IoT.

G: Resource definition

A list of local resources and machine learning resources on the AWS Greengrass core, with associated configuration data. For more information, see Access Local Resources with Lambda Functions (p. 129) and Perform Machine Learning Inference (p. 154).

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

Devices in AWS Greengrass

There are two types of devices:
• AWS Greengrass cores.
• AWS IoT devices connected to an AWS Greengrass core.

An AWS Greengrass core is an AWS IoT device that runs specialized AWS Greengrass software that communicates directly with the AWS IoT and AWS Greengrass cloud services. It is an AWS IoT device with its own certificate used for authenticating with AWS IoT. It has a device shadow and exists in the AWS IoT device registry. AWS Greengrass cores run a local Lambda runtime, a deployment agent, and an IP address tracker that sends IP address information to the AWS Greengrass cloud service to allow AWS IoT devices to automatically discover their group and core connection information.
Any AWS IoT device can connect to an AWS Greengrass core. These devices run software written with the AWS IoT Device SDK.

The following table shows how these device types are related.

<table>
<thead>
<tr>
<th>Core</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>Certificate</td>
<td></td>
</tr>
<tr>
<td>IoT Policy</td>
<td></td>
</tr>
<tr>
<td>IoT Thing</td>
<td></td>
</tr>
<tr>
<td>Device use sample</td>
<td>Gateway</td>
</tr>
<tr>
<td>Software</td>
<td>Sensor and/or Actuator</td>
</tr>
<tr>
<td>Group membership</td>
<td></td>
</tr>
<tr>
<td>Functions outside an AWS</td>
<td></td>
</tr>
<tr>
<td>Greengrass Group</td>
<td></td>
</tr>
</tbody>
</table>

The AWS 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 Greengrass services.
- MQTT core server certificate in /greengrass/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 SDKs are used when working with AWS Greengrass:

GGC 1.6.0

AWS SDKs

Using the AWS SDKs, you can build applications that work with any AWS service, including Amazon S3, Amazon DynamoDB, AWS IoT, AWS Greengrass, and more. In the context of AWS Greengrass, you can use the AWS SDK in deployed Lambda functions to make direct calls to any AWS service. For more information, see the section called “SDKs” (p. 122).
AWS IoT Device SDKs

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

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

AWS Greengrass Core SDK

The AWS Greengrass Core SDK enables Lambda functions to interact with the AWS Greengrass core on which they run in order to publish messages, interact with the local Device Shadow service, or invoke other deployed Lambda functions. This SDK is used exclusively for writing Lambda functions running in the Lambda runtime on an AWS Greengrass core. For more information, see the section called “SDKs” (p. 122).

GGC v1.5.0

AWS SDKs

Using the AWS SDKs, you can build applications that work with any AWS service, including Amazon S3, Amazon DynamoDB, AWS IoT, AWS Greengrass, and more. In the context of AWS Greengrass, you can use the AWS SDK inside deployed Lambda functions to make direct calls to any AWS service.

AWS IoT Device SDKs

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

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

AWS Greengrass Core SDK

The AWS Greengrass Core SDK enables Lambda functions to interact with the AWS Greengrass core on which they run in order to publish messages, interact with the local Device Shadow service, or invoke other deployed Lambda functions. This SDK is used exclusively for writing Lambda functions running in the Lambda runtime on an AWS Greengrass core. Lambda functions running on an AWS Greengrass core can interact with AWS cloud services directly using the AWS SDK. The AWS Greengrass Core SDK and the AWS SDK are contained in different packages, so you can use both packages simultaneously. You can download the AWS Greengrass Core SDK from the Software page of the AWS IoT console.

The AWS Greengrass Core SDK follows the AWS SDK programming model. It allows you to easily port Lambda functions developed for the cloud to Lambda functions that run on an AWS Greengrass core. For example, using the AWS SDK, the following Lambda function publishes 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 this Lambda function for execution on an AWS Greengrass core, replace the `import boto3` statement with the `import greengrasssdk`, as shown in the following snippet:

**Note**
The AWS Greengrass Core SDK only supports sending MQTT messages with QoS = 0.

```python
import greengrasssdk
client = greengrasssdk.client('iot-data')
response = client.publish(
    topic='/some/topic',
    qos=0,
    payload='some payload'.encode()
)
```

This allows you to test your Lambda functions in the cloud and migrate them to AWS Greengrass with minimal effort.

**Note**
The AWS SDK is natively part of the environment of a Lambda function that runs in the AWS cloud. If you want to use `boto3` in a Lambda function that's deployed to an AWS Greengrass core, make sure to include the AWS SDK in your package. In addition, if you use both the AWS Greengrass Core SDK and the AWS SDK in the same package, make sure that your Lambda functions use the correct namespaces. For more information about how to create your deployment package, see:

- Creating a Deployment Package (Python)
- Creating a Deployment Package (Node.js)
- Creating a Deployment Package (Java)

GGC v1.3.0

**AWS SDKs**

Using the AWS SDKs, you can build applications that work with any AWS service, including Amazon S3, Amazon DynamoDB, AWS IoT, AWS Greengrass, and more. In the context of AWS Greengrass, you can use the AWS SDK inside deployed Lambda functions to make direct calls to any AWS service.

**AWS IoT Device SDKs**

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

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

**AWS Greengrass Core SDK**

The AWS Greengrass Core SDK enables Lambda functions to interact with the AWS Greengrass core on which they run in order to publish messages, interact with the local Device Shadow service, or invoke other deployed Lambda functions. This SDK is used exclusively for writing Lambda functions running in the Lambda runtime on an AWS Greengrass core. Lambda
functions running on an AWS Greengrass core can interact with AWS cloud services directly using the AWS SDK. The AWS Greengrass Core SDK and the AWS SDK are contained in different packages, so you can use both packages simultaneously. You can download the AWS Greengrass Core SDK from the Software page of the AWS IoT console.

The AWS Greengrass Core SDK follows the AWS SDK programming model. It allows you to easily port Lambda functions developed for the cloud to Lambda functions that run on an AWS Greengrass core. For example, using the AWS SDK, the following Lambda function publishes 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 this Lambda function for execution on an AWS Greengrass core, replace the `import boto3` statement with the `import greengrasssdk`, as shown in the following snippet:

**Note**
The AWS Greengrass Core SDK only supports sending MQTT messages with QoS = 0.

```python
import greengrasssdk
client = greengrasssdk.client('iot-data')
response = client.publish(topic='/some/topic',
qos=0,
payload='some payload'.encode())
```

This allows you to test your Lambda functions in the cloud and migrate them to AWS Greengrass with minimal effort.

**Note**
The AWS SDK is natively part of the environment of a Lambda function that runs in the AWS cloud. If you want to use `boto3` in a Lambda function that's deployed to an AWS Greengrass core, make sure to include the AWS SDK in your package. In addition, if you use both the AWS Greengrass Core SDK and the AWS SDK in the same package, make sure that your Lambda functions use the correct namespaces. For more information about how to create your deployment package, see:

- Creating a Deployment Package (Python)
- Creating a Deployment Package (Node.js)
- Creating a Deployment Package (Java)

GGC v1.1.0

AWS SDKs

Using the AWS SDKs, you can build applications that work with any AWS service, including Amazon S3, Amazon DynamoDB, AWS IoT, AWS Greengrass, and more. In the context of AWS Greengrass, you can use the AWS SDK inside deployed Lambda functions to make direct calls to any AWS service.
AWS IoT Device SDKs

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

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

AWS Greengrass Core SDK

The AWS Greengrass Core SDK enables Lambda functions to interact with the AWS Greengrass core on which they run in order to publish messages, interact with the local Device Shadow service, or invoke other deployed Lambda functions. This SDK is used exclusively for writing Lambda functions running in the Lambda runtime on an AWS Greengrass core. Lambda functions running on an AWS Greengrass core can interact with AWS cloud services directly using the AWS SDK. The AWS Greengrass Core SDK and the AWS SDK are contained in different packages, so you can use both packages simultaneously. You can download the AWS Greengrass Core SDK from the Software page of the AWS IoT console.

The AWS Greengrass Core SDK follows the AWS SDK programming model. It allows you to easily port Lambda functions developed for the cloud to Lambda functions that run on an AWS Greengrass core. For example, using the AWS SDK, the following Lambda function publishes 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 this Lambda function for execution on an AWS Greengrass core, replace the import boto3 statement with the import greengrasssdk, as shown in the following snippet:

**Note**
The AWS Greengrass Core SDK only supports sending MQTT messages with QoS = 0.

```python
import greengrasssdk
client = greengrasssdk.client('iot-data')
response = client.publish(
    topic='some/topic',
    qos=0,
    payload='some payload'.encode()
)
```

This allows you to test your Lambda functions in the cloud and migrate them to AWS Greengrass with minimal effort.

**Note**
The AWS SDK is natively part of the environment of a Lambda function that runs in the AWS cloud. If you want to use boto3 in a Lambda function that's deployed to an AWS Greengrass core, make sure to include the AWS SDK in your package. In addition, if you use both the AWS Greengrass Core SDK and the AWS SDK in the same package, make sure that
your Lambda functions use the correct namespaces. For more information about how to create your deployment package, see:

- Creating a Deployment Package (Python)
- Creating a Deployment Package (Node.js)
- Creating a Deployment Package (Java)

GGC v1.0.0

AWS SDKs

Using the AWS SDKs, you can build applications that work with any AWS service, including Amazon S3, Amazon DynamoDB, AWS IoT, AWS Greengrass, and more. In the context of AWS Greengrass, you can use the AWS SDK inside deployed Lambda functions to make direct calls to any AWS service.

AWS IoT Device SDKs

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

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

AWS Greengrass Core SDK

The AWS Greengrass Core SDK enables Lambda functions to interact with the AWS Greengrass core on which they run in order to publish messages, interact with the local Device Shadow service, or invoke other deployed Lambda functions. This SDK is used exclusively for writing Lambda functions running in the Lambda runtime on an AWS Greengrass core. Lambda functions running on an AWS Greengrass core can interact with AWS cloud services directly using the AWS SDK. The AWS Greengrass Core SDK and the AWS SDK are contained in different packages, so you can use both packages simultaneously. You can download the AWS Greengrass Core SDK from the Software page of the AWS IoT console.

The AWS Greengrass Core SDK follows the AWS SDK programming model. It allows you to easily port Lambda functions developed for the cloud to Lambda functions that run on an AWS Greengrass core. For example, using the AWS SDK, the following Lambda function publishes 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 this Lambda function for execution on an AWS Greengrass core, replace the `import boto3` statement with the `import greengrasssdk`, as shown in the following snippet:

```python
Note
The AWS Greengrass Core SDK only supports sending MQTT messages with QoS = 0.

import greengrasssdk
```
This allows you to test your Lambda functions in the cloud and migrate them to AWS Greengrass with minimal effort.

**Note**
The AWS SDK is natively part of the environment of a Lambda function that runs in the AWS cloud. If you want to use `boto3` in a Lambda function that’s deployed to an AWS Greengrass core, make sure to include the AWS SDK in your package. In addition, if you use both the AWS Greengrass Core SDK and the AWS SDK in the same package, make sure that your Lambda functions use the correct namespaces. For more information about how to create your deployment package, see:

- Creating a Deployment Package (Python)

---

### Supported Platforms and Requirements

The AWS Greengrass core software is supported on the platforms listed here, and requires a few dependencies.

**GGC v1.6.0**

- **Supported platforms:**
  - Architecture: ARMv7L; OS: Linux; Distribution: Raspbian Jessie, 2017-03-02
  - Architecture: x86_64; OS: Linux; Distribution: Amazon Linux (amzn-ami-hvm-2016.09.1.20170119-x86_64-ebc)
  - Architecture: x86_64; OS: Linux; Distribution: Ubuntu 14.04 – 16.04
  - Architecture: ARMv8 (AArch64); OS: Linux; Distribution: Ubuntu 14.04 – 16.04 (Annapurna Alpine V2)

- **The following items are required:**
  - Minimum 128 MB RAM allocated to the AWS Greengrass core device.
  - Linux kernel version 4.4 or greater: while several versions may work with AWS Greengrass, for optimal security and performance, we recommend version 4.4 or greater.
  - **Glibc library** version 2.14 or greater.
  - The `/var/run` directory must be present on the device.
  - AWS Greengrass requires hardlink and softlink protection to be enabled on the device. Without this, AWS 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_UTS_NS, CONFIG_USER_NS, CONFIG_PID_NS
    - CGroups: CONFIG_CGROUP_DEVICE, CONFIG_CGROUPS, CONFIG_MEMCG
    - Others: CONFIG_POSIX_MQUEUE, CONFIG_OVERLAY_FS, CONFIG_HAVE_ARCH_SECCOMP_FILTER, CONFIG_SECCOMP_FILTER, CONFIG_KEYS, CONFIG_SECCOMP
  - `/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 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. 129) are used to open files on the AWS 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.

- **OpenSSL** 1.01 or greater is required for Greengrass OTA Agent (p. 115) as well as the following commands: `wget`, `realpath`, `tar`, `readlink`, `basename`, `dirname`, `pidof`, `df`, `grep`, and `umount`.

**GGC v1.5.0**

- Supported platforms:
  - Architecture: ARMv7l; OS: Linux; Distribution: Raspbian Jessie, 2017-03-02
  - Architecture: x86_64; OS: Linux; Distribution: Amazon Linux (amzn-ami-hvm-2016.09.1.20170119-x86_64-ecs)
  - Architecture: x86_64; OS: Linux; Distribution: Ubuntu 14.04 – 16.04
  - Architecture: ARMv8 (AArch64); OS: Linux; Distribution: Ubuntu 14.04 – 16.04 (Annapurna Alpine V2)

- The following items are required:
  - Minimum 128 MB RAM allocated to the AWS Greengrass core device.
  - Linux kernel version 4.4 or greater: while several versions may work with AWS Greengrass, for optimal security and performance, we recommend version 4.4 or greater.
  - **Glibc library** version 2.14 or greater.
  - The `/var/run` directory must be present on the device.
  - AWS Greengrass requires hardlink and softlink protection to be enabled on the device. Without this, AWS 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_UTS_NS`, `CONFIG_USER_NS`, `CONFIG_PID_NS`
    - CGroups: `CONFIG_CGROUP_DEVICE`, `CONFIG_CGROUPS`, `CONFIG_MEMCG`
    - Others: `CONFIG_POSIX_MQUEUE`, `CONFIG_OVERLAY_FS`, `CONFIG_HAVE_ARCH_SECCOMP_FILTER`, `CONFIG_SECCOMP_FILTER`, `CONFIG_KEYS`, `CONFIG_SECCOMP`
    - `/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 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. 129) are used to open files on the AWS 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.

• OpenSSL 1.01 or greater is required for Greengrass OTA Agent (p. 115) as well as the following commands: `wget`, `realpath`, `tar`, `readlink`, `basename`, `dirname`, `pidof`, `df`, `grep`, and `umount`.

GGC v1.3.0

• Supported platforms:
  - Architecture: ARMv7l; OS: Linux; Distribution: Raspbian Jessie, 2017-03-02
  - Architecture: x86_64; OS: Linux; Distribution: Amazon Linux (amzn-ami-hvm-2016.09.1.20170119-x86_64-ebs)
  - Architecture: x86_64; OS: Linux; Distribution: Ubuntu 14.04 – 16.04
  - Architecture: ARMv8 (AArch64); OS: Linux; Distribution: Ubuntu 14.04 – 16.04 (Annapurna Alpine V2)

• The following items are required:
  - Minimum 128 MB RAM allocated to the AWS Greengrass core device.
  - Linux kernel version 4.4 or later. Although several versions might work with AWS Greengrass, for optimal security and performance, we recommend version 4.4 or later.
  - **Glibc library** version 2.14 or later.
  - The `/var/run` directory must be present on the device.
  - AWS Greengrass requires hardlink and softlink protection to be enabled on the device. Without this, AWS 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_UTS_NS`, `CONFIG_USER_NS`, `CONFIG_PID_NS`
    - CGroups: `CONFIG_CGROUP_DEVICE`, `CONFIG_CGROUPS`, `CONFIG_MEMCG`
    - Others: `CONFIG_POSIX_MQUEUE`, `CONFIG_OVERLAY_FS`, `CONFIG_HAVE_ARCH_SECCOMP_FILTER`, `CONFIG_SECCOMP_FILTER`, `CONFIG_KEYS`, `CONFIG_SECCOMP`
  - The `sqlite3` package is required for AWS IoT device shadows. Make sure it's added to your `PATH` environment variable.
  - `/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 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 might be optional:
  - The `devices cgroup` must be enabled and mounted if Lambda functions with local resource access (LRA) (p. 129) are used to open files on the AWS Greengrass core device.
  - **Python** version 2.7 is required if Python Lambda functions are used. If so, make sure that it's added to your `PATH` environment variable.
  - **Node.js** version 6.10 or later is required if Node.js Lambda functions are used. If so, make sure that it's added to your `PATH` environment variable.
• **Java** version 8 or later is required if Java Lambda functions are used. If so, make sure that it’s added to your `PATH` environment variable.

• **OpenSSL** 1.01 or later is required for Greengrass OTA Agent (p. 115) as are the following commands: `wget`, `realpath`, `tar`, `readlink`, `basename`, `dirname`, `pidof`, `df`, `grep`, and `umount`.

**GGC v1.1.0**

• Supported platforms:
  - Architecture: ARMv7l; OS: Linux; Distribution: Raspbian Jessie, 2017-03-02
  - Architecture: x86_64; OS: Linux; Distribution: Amazon Linux (amzn-ami-hvm-2016.09.1.20170119-x86_64-ebs)
  - Architecture: x86_64; OS: Linux; Distribution: Ubuntu 14.04 – 16.04
  - Architecture: ARMv8 (AArch64); OS: Linux; Distribution: Ubuntu 14.04 – 16.04 (Annapurna Alpine V2)

  The following items are required:
  - Minimum 128 MB RAM allocated to the AWS Greengrass core device.
  - Linux kernel version 4.4 or later. Although several versions might work with AWS Greengrass, for optimal security and performance, we recommend version 4.4 or later.
  - **Glibc library** version 2.14 or later.
  - The `/var/run` directory must be present on the device.
  - AWS Greengrass requires hardlink and softlink protection to be enabled on the device. Without this, AWS 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`
  - The `sqlite3` package is required for AWS IoT device shadows. Make sure it’s added to your `PATH` environment variable.
  - `/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 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 might be optional:
  - **Python** version 2.7 is required if Python Lambda functions are used. If so, make sure that it’s added to your `PATH` environment variable.
  - **Node.js** version 6.10 or later is required if Node.js Lambda functions are used. If so, make sure that it’s added to your `PATH` environment variable.
  - **Java** version 8 or later is required if Java Lambda functions are used. If so, make sure that it’s added to your `PATH` environment variable.

**GGC v1.0.0**

• Supported platforms:
  - Architecture: ARMv7l; OS: Linux; Distribution: Raspbian Jessie, 2017-03-02
- Architecture: x86_64; OS: Linux; Distribution: Amazon Linux (amzn-ami-hvm-2016.09.1.20170119-x86_64-ebs)
- Architecture: x86_64; OS: Linux; Distribution: Ubuntu 14.04 – 16.04
- Architecture: ARMv8 (AArch64); OS: Linux; Distribution: Ubuntu 14.04 – 16.04 (Annapurna Alpine V2)

The following items are required:
- Minimum 128 MB RAM allocated to the AWS Greengrass core device.
- Linux kernel version 4.4 or later. Although several versions might work with AWS Greengrass, for optimal security and performance, we recommend version 4.4 or later.
- Glibc library version 2.14 or later.
- The `/var/run` directory must be present on the device.
- AWS Greengrass requires hardlink and softlink protection to be enabled on the device. Without this, AWS 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_UTS_NS, CONFIG_USER_NS, CONFIG_PID_NS
  - CGroups: CONFIG_CGROUP_DEVICE, CONFIG_CGROUPS, CONFIG_MEMCG
  - Others: CONFIG_POSIX_MQUEUE, CONFIG_OVERLAY_FS, CONFIG_HAVE_ARCH_SECCOMP_FILTER, CONFIG_SECCOMP_FILTER, CONFIG_KEYS, CONFIG_SECCOMP
- The `sqlite3` package is required for AWS IoT device shadows. Make sure it's added to your `PATH` environment variable.
- `/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 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 might be optional:
  - Python version 2.7 is required if Python Lambda functions are used. If so, make sure that it's added to your `PATH` environment variable.

We Want to Hear from You

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

Configure the AWS Greengrass Core

An AWS Greengrass core is an AWS IoT thing. 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 Greengrass core software, which enables it to manage local processes for Greengrass groups, such as communication, shadow sync, and token exchange.

The AWS Greengrass core software consists of:
- A message manager that routes messages between devices, Lambda functions, and AWS IoT.
- A Lambda runtime that runs user-defined Lambda functions.
- An implementation of the Device Shadow service that provides a local copy of shadows, which represent your devices. Shadows can be configured to sync with the cloud.
• A deployment agent that is notified of new or updated AWS Greengrass group configuration. When a new or updated configuration is detected, the deployment agent downloads the configuration data and restarts the AWS Greengrass core.

**Note**
greengrass-root represents the path to where the AWS Greengrass core software is installed on your device. If you installed the software by following the Getting Started (p. 29) tutorial, then this is the /greengrass directory.

### AWS Greengrass Core Configuration File

The configuration file for the AWS Greengrass core software is the config.json file, which is located in the /greengrass-root/config directory. You can review the contents of this file by running the following command:

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

**Note**
In AWS Greengrass Core v1.0.0, config.json is deployed to /greengrass-root/configuration.

The following is an example config.json file.

GGC v1.6.0

```json
{
    "coreThing": {
        "caPath": "root-ca-pem",
        "certPath": "cloud-pem.crt",
        "keyPath": "cloud-pem-key",
        "thingArn": "thing-arn",
        "iotHost": "host-prefix.iot.aws-region.amazonaws.com",
        "ggHost": "greengrass.iot.aws-region.amazonaws.com",
        "keepAlive": 600,
        "mqttMaxConnectionRetryInterval": 60
    },
    "runtime": {
        "cgroup": {
            "useSystemd": "yes"
        },
        "managedRespawn": true,
        "writeDirectory": "/write-directory"
    }
}
```

**Note**
If you use the Easy group creation option from the AWS 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 /greengrass-root/certs directory.</td>
<td>Save the file under /greengrass-root/certs.</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>------------</td>
<td>------------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>certPath</td>
<td>The path to the AWS Greengrass core certificate relative to /greengrass-root/certs directory.</td>
<td>Save the file under /greengrass-root/certs.</td>
</tr>
<tr>
<td>keyPath</td>
<td>The path to the AWS Greengrass core private key relative to /greengrass/certs directory.</td>
<td>Save the file under /greengrass-root/certs.</td>
</tr>
<tr>
<td>thingArn</td>
<td>The Amazon Resource Name (ARN) of the AWS IoT thing that represents the AWS Greengrass core device.</td>
<td>Find this for your core in the AWS Greengrass console under Cores, or by running the aws greengrass list-core-definitions 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 aws iot describe-endpoint CLI command.</td>
</tr>
<tr>
<td>ggHost</td>
<td>Your AWS Greengrass endpoint.</td>
<td>This value uses the format greengrass.iot.region.amazonaws.com. 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 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 yes or no. Run the check_ggc_dependencies script in Module 1 (p. 30) 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 true or false. For more information, see OTA Updates of AWS Greengrass Core Software (p. 114).</td>
</tr>
<tr>
<td>writeDirectory</td>
<td>The write directory where AWS Greengrass creates all read-write resources.</td>
<td>This is an optional value. For more information, see Configure a Write Directory for AWS Greengrass (p. 23).</td>
</tr>
</tbody>
</table>

GGC v1.5.0

```json
{
```
The `config.json` file appears in `/greengrass/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/certs</code> folder.</td>
<td>Save the file under the <code>/greengrass/certs</code> folder.</td>
</tr>
<tr>
<td>certPath</td>
<td>The path to the AWS Greengrass core certificate relative to the <code>/greengrass/certs</code> folder.</td>
<td>Save the file under the <code>/greengrass/certs</code> folder.</td>
</tr>
<tr>
<td>keyPath</td>
<td>The path to the AWS Greengrass core private key relative to the <code>/greengrass/certs</code> folder.</td>
<td>Save the file under the <code>/greengrass/certs</code> folder.</td>
</tr>
<tr>
<td>thingArn</td>
<td>The Amazon Resource Name (ARN) of the AWS IoT thing that represents the AWS Greengrass core.</td>
<td>Find this for your core in the AWS Greengrass console under <code>Cores</code>, or by running the <code>aws greengrass list-core-definitions</code> 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 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>Values are <code>yes</code> or <code>no</code>. Use the dependency script in Module 1 (p. 30) to see if your device uses <code>systemd</code>.</td>
</tr>
</tbody>
</table>
### Field | Description | Notes
--- | --- | ---
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 Greengrass Core Software](p. 114).

**GGC v1.3.0**

```json
{
    "coreThing": {
        "caPath": "ROOT_CA_PEM_HERE",
        "certPath": "CLOUD_PEM_CRT_HERE",
        "keyPath": "CLOUD_PEM_KEY_HERE",
        "thingArn": "THING_ARN_HERE",
        "iotHost": "HOST_PREFIX_HERE.iot.AWS_REGION_HERE.amazonaws.com",
        "ggHost": "greengrass.iot.AWS_REGION_HERE.amazonaws.com",
        "keepAlive": 600
    },
    "runtime": {
        "cgroup": {
            "useSystemd": "yes|no"
        }
    },
    "managedRespawn": true
}
```

The `config.json` file appears in `/greengrass/config/` and contains the following parameters:

### Field | Description | Notes
--- | --- | ---
caPath | The path to the AWS IoT root CA relative to the `/greengrass/certs` folder. | Save the file under the `/greengrass/certs` folder.
certPath | The path to the AWS Greengrass core certificate relative to the `/greengrass/certs` folder. | Save the file under the `/greengrass/certs` folder.
keyPath | The path to the AWS Greengrass core private key relative to `/greengrass/certs` folder. | Save the file under the `/greengrass/certs` folder.
thingArn | The Amazon Resource Name (ARN) of the AWS IoT thing that represents the AWS Greengrass core. | You can find it in the AWS Greengrass console under the definition for your AWS IoT thing.
iotHost | Your AWS IoT endpoint. | You can find it in the AWS IoT console under **Settings**.
ggHost | Your AWS Greengrass endpoint. | You can find it in the AWS IoT console under **Settings** with `greengrass` prepended.
### AWS Greengrass Core Configuration File

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<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. 30) 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>For more information, see OTA Updates of AWS Greengrass Core Software (p. 114).</td>
</tr>
</tbody>
</table>

**GGC v1.1.0**

```json
{
  "coreThing": {
    "caPath": "ROOT_CA_PEM_HERE",
    "certPath": "CLOUD_PEM_CRT_HERE",
    "keyPath": "CLOUD_PEM_KEY_HERE",
    "thingArn": "THING_ARN_HERE",
    "iotHost": "HOST_PREFIX_HERE.iot.AWS_REGION_HERE.amazonaws.com",
    "ggHost": "greengrass.iot.AWS_REGION_HERE.amazonaws.com",
    "keepAlive": 600
  },
  "runtime": {
    "cgroup": {
      "useSystemd": "yes/no"
    }
  }
}
```

The `config.json` file exists in `/greengrass/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/certs</code> folder.</td>
<td>Save the file under the <code>/greengrass/certs</code> folder.</td>
</tr>
<tr>
<td>certPath</td>
<td>The path to the AWS Greengrass core certificate relative to the <code>/greengrass/certs</code> folder.</td>
<td>Save the file under the <code>/greengrass/certs</code> folder.</td>
</tr>
<tr>
<td>keyPath</td>
<td>The path to the AWS Greengrass core private key relative to the <code>/greengrass/certs</code> folder.</td>
<td>Save the file under the <code>/greengrass/certs</code> folder.</td>
</tr>
<tr>
<td>thingArn</td>
<td>The Amazon Resource Name (ARN) of the AWS IoT thing that represents the AWS Greengrass core.</td>
<td>You can find it in the AWS Greengrass console under the definition for your AWS IoT thing.</td>
</tr>
</tbody>
</table>
## AWS Greengrass Core Configuration File

### Field | Description | Notes
--- | --- | ---
`iotHost` | Your AWS IoT endpoint. | You can find it in the AWS IoT console under **Settings**.  
`ggHost` | Your AWS Greengrass endpoint. | You can find it in the AWS IoT console under **Settings** with `greengrass` prepended.  
`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. 30) to see if your device uses `systemd`.  

### GGC v1.0.0

```json
{
    "coreThing": {
        "caPath": "ROOT_CA_PEM_HERE",
        "certPath": "CLOUD_PEM_CRT_HERE",
        "keyPath": "CLOUD_PEM_KEY_HERE",
        "thingArn": "THING_ARN_HERE",
        "iotHost": "HOST_PREFIX_HERE.iot.AWS_REGION_HERE.amazonaws.com",
        "ggHost": "greengrass.iot.AWS_REGION_HERE.amazonaws.com",
        "keepAlive": 600
    },
    "runtime": {
        "cgroup": {
            "useSystemd": "yes|no"
        }
    }
}
```

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

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

---

22
### Configure a Write Directory for AWS Greengrass

This feature is available for AWS Greengrass Core v1.6.0 only.

By default, the AWS Greengrass core software is deployed under a single root directory where AWS Greengrass performs all read and write operations. However, you can configure AWS Greengrass to use a separate directory for all write operations, including creating directories and files. In this case, AWS 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 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/
  |           |-- daemon
  |       |-- greengrassd
  |       |-- lambda/
  |       |-- LICENSE/
  |       |-- release_notes_package-version.html
```
To Configure a Write Directory

1. Run the following command to stop the AWS 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 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. 25).

   Otherwise, start the AWS Greengrass daemon:

   ```
   cd /greengrass-root/ggc/core/
   ```
To Make the Greengrass Root Directory Read-Only

1. Grant required access to the AWS Greengrass:
   - **Note**
     This step is required only if you want to make the Greengrass root directory read-only. The write directory must be configured before you begin this procedure.
     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/1.6.0/lambda/
        ```

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 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 Agent" (p. 115) and the section called “AWS Greengrass Core Update with Managed Respawn” (p. 118).

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.

### 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 Greengrass manages the queue by using a system-defined GGCloudSpooler Lambda function.

### Configure the MQTT Message Queue

This feature is available for AWS Greengrass Core v1.6.0 only.

You can configure AWS 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 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 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 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 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.

Note
This procedure assumes that you're updating the configuration of the latest group version of an existing group.

1. Get the IDs of the target Greengrass group and group version.

```
aws greengrass list-groups
```

2. Copy the Id and LatestVersion properties of your target group from 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 `ComponentDefinitionVersionArn` for each group component 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.
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_SIZE_BYTES` that's greater than or equal to 262144. A smaller size prevents the AWS Greengrass core software from starting.

```bash
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_SIZE_BYTES": "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-defined 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 by using the `ComponentDefinitionVersionArn` values 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.

```bash
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.
    - 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.
To update the storage settings, you use the AWS 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-defined 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).

**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. 30). Then to use systemd, make sure that the `useSystemd` parameter in `config.json` (p. 17) 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.
Getting Started with AWS Greengrass

This tutorial includes six modules, each designed to show you AWS Greengrass basics and help you get started in as few steps as possible. This tutorial covers:

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

Requirements

To complete this tutorial, you will need the following:

- A Mac, Windows PC, or UNIX-like system.
- An Amazon Web Services (AWS) account. If you don’t have an AWS account, see the section called “Create an AWS Account” (p. 30).
- The use of an AWS region that supports AWS Greengrass. For the list of supported regions for AWS Greengrass, see AWS Regions and Endpoints in the AWS General Reference.
  Important
  Make note of your region to ensure that it is consistently used throughout this tutorial – inadvertently switching regions midway through the tutorial would be problematic. Note that the last exercise in this tutorial assumes the US East (N. Virginia) region, so you may want to only use the US East (N. Virginia) region, as possible.
- A Raspberry Pi 3 Model B with a 8 GB microSD card, or an Amazon EC2 instance. Because AWS Greengrass is intended to be used with physical hardware, we recommend that you use a Raspberry Pi.
  Note
  If the model of your Raspberry Pi is unknown, you can run the following command:

  cat /proc/cpuinfo

  Near the bottom of the listing, note the value of the Revision attribute. You can determine the model of your Pi by using this value along with the table at Which Pi have I got? For example, if the value of Revision is a02082, then from the table we see that the Pi is a 3 Model B. Additionally, the architecture of your Pi must be armv71 or greater. To determine the architecture of your Raspberry Pi, run the following command:

  uname -m

  The result must be greater than or equal to armv71.
- Basic familiarity with Python 2.7.

Although this tutorial focuses on running AWS Greengrass on a Raspberry Pi or an Amazon EC2 instance, other platforms are supported. For more information, see Supported Platforms and Requirements (p. 12).
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
     Ensure that your account has administrative privileges before proceeding.

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 Greengrass.

- Important
  Use the Filter View drop-down list in the upper-right corner of this webpage to choose your platform.

This module should take less than 30 minutes to complete.

Setting Up a Raspberry Pi

If you are setting up a Raspberry Pi for the first time, you must follow all of these steps. If you are using an existing Raspberry Pi, 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 quick format the SD card.

2. Download the Raspbian Jessie operating system as a .zip file. Only 2017-03-02-raspbian-jessie.zip is currently supported by AWS Greengrass.

3. Using an SD card-writing tool (such as Etcher), follow the tool's instructions to flash the downloaded 2017-03-02-raspbian-jessie.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 may want to configure the Pi's keyboard layout before proceeding. To do so, choose the Raspberry icon in the upper-right, choose Preferences and then Mouse and Keyboard Settings. Next, choose the Keyboard tab, 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 proceeding. If you’re in a work environment or behind a firewall, you may need to connect your Pi and
your computer to the guest network in order to get both devices on the same network. This approach, however, may disconnect your computer from local network resources such as your intranet. One solution may be to connect the Pi to the guest Wi-Fi network, 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 via 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:

![Interfacing Options](image)

Scroll down and choose **Interfacing Options** and then choose **P2 SSH**. When prompted, choose **Yes** using the **Tab** key (followed by **Enter**). SSH should now be enabled, choose **OK**. **Tab** key to **Finish** and then press the **Enter** key. Lastly, reboot your Pi by running 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 related message *Are you sure you want to continue connecting (yes/no)?*, enter yes. Additionally, the default password for the Raspberry Pi is **raspberry**.

If you are using macOS, open a Terminal window and type the following:

```
ssh pi@IP-address
```

Here, **IP-address** corresponds to the IP address of your Raspberry Pi that you obtained by using the prior `hostname -I` command.

If you are using Windows, you need to install and configure PuTTY. Choose **Connection**, **Data**, and make sure that **Prompt** is selected:
Next, choose **Session**, type the IP address of the Raspberry Pi, and choose **Open** using default settings. For example (your IP address, in all likelihood, will be different):
If a PuTTY Security Alert dialog is displayed, choose an appropriate response such as Yes.

This will result in a terminal window similar to the following. The default Raspberry Pi login and password are **pi** and **raspberry**, respectively.
Note
If your computer is connected to a remote network using VPN (such as a work related network), this may cause difficulty connecting from the computer to the Raspberry Pi using SSH.

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

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

10. Run the following commands to update the Linux kernel version of your Raspberry Pi.

```
sudo apt-get install rpi-update
sudo rpi-update b81a11258fc911170b40a0b09bbd63c84bc5ad59
```

Although several kernel versions might work with AWS Greengrass, for the best security and performance, we recommend that you use the kernel version indicated in step 2. In order to activate the new firmware, reboot your Raspberry Pi:

```
sudo reboot
```

As applicable, reconnect to the Raspberry Pi using SSH after minute or so. Next, run the following command to ensure you have the correct kernel version:

```
uname -a
```

You should receive output similar to the following, the key item being the Linux Raspberry Pi version information 4.9.30:
11. To improve security on the Pi device, run the following commands to enable hardlink and softlink protection at operating system start-up.

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

If you see the `98-rpi.conf` file, use a text editor (such as `leafpad`, `nano`, or `vi`) to add the following two lines to the end of the file (you can run the text editor using the `sudo` command to avoid write permission issues, as in `sudo nano 98-rpi.conf`).

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

If you do not see the `98-rpi.conf` file, follow the instructions in the `README.sysctl` file.

Now reboot the Pi:

```
    sudo reboot
```

After about a minute, connect to the Pi using SSH as applicable and then run the following commands from a Raspberry Pi terminal to confirm the hardlink/symlink change:

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

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

12. Your Raspberry Pi should now be ready for AWS Greengrass. To ensure that you have all of the dependencies required for AWS Greengrass, download the AWS Greengrass dependency checker from the GitHub repository and run it on the Pi as follows:

```
    cd /home/pi/Downloads
    git clone https://github.com/aws-samples/aws-greengrass-samples.git
    cd awa-greengrass-samples
    cd greengrass-dep-co-checker-GGCv1.6.0
    sudo modprobe configs
    sudo ./check_ggc_dependencies | more
```

With respect to the `more` command, press the `Spacebar` key to display another screen of text.

**Important**

Because this tutorial only uses the AWS IoT Device SDK for Python, you can ignore warnings about the missing optional NodeJS 6.10 and Java 8 prerequisites that the `check_ggc_dependencies` script may produce.

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

Your Raspberry Pi configuration is complete. Continue to the section called "Module 2: Installing the Greengrass Core Software" (p. 43).

### Setting Up an Amazon EC2 Instance

1. Sign in to the [AWS Management Console](https://aws.amazon.com/console/) and launch an Amazon EC2 instance using an Amazon Linux AMI (Amazon Machine Image). For information about Amazon EC2 instances, see the [Amazon EC2 Getting Started Guide](https://aws.amazon.com/ec2/quick-start/).
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 Greengrass core. In the left pane of the Amazon EC2 console, choose **Security Groups**.

Choose the instance that you just launched, and then choose the **Inbound** tab.
By default, only one port for SSH is enabled. To enable port 8883, choose the Edit button. Next, choose the Add Rule button and create a custom TCP rule as shown below, then choose Save.

3. In the left pane, choose Instances, choose your instance, and then choose the Connect button. Connect to your Amazon EC2 instance by using SSH. You can use PuTTY for Windows or Terminal for macOS.
4. Once connected to your Amazon EC2 instance through SSH, run the following commands to create user ggc_user and group ggc_group:

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

5. To improve security on the device, enable hardlink/softlink protection on the operating system at start-up. To do so, run the following commands:

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

Using your favorite text editor (leappad, nano, vi, etc.), add the following two lines to the end of the `00-defaults.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`).

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

Run the following command to reboot the Amazon EC2 instance.
After a few minutes, connect to your instance by using SSH as above. 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 is an AWS Greengrass dependency:

```bash
curl https://raw.githubusercontent.com/tianon/cgroupfs-mount/951c38ee8d80233045b0dede20d85ec1c0f8ed312/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 Greengrass. To be sure that you have all of the dependencies, extract and run the following AWS Greengrass dependency script from the GitHub repository:

```bash
sudo yum install git
git clone https://github.com/aws-samples/aws-greengrass-samples.git
cd aws-greengrass-samples
cd greengrass-dependency-checker-GGCv1.6.0
sudo ./check_ggc_dependencies
```

Your Amazon EC2 instance configuration is complete. Continue to the section called “Module 2: Installing the Greengrass Core Software” (p. 43).

### Setting Up Other Devices

If you are new to AWS Greengrass, we recommend that you use a Raspberry Pi or an Amazon EC2 instance and follow the steps provided above to set up the device. Follow the below steps to make your own AWS Greengrass-supported device ready for AWS Greengrass.

1. To make sure you have other devices ready to run AWS Greengrass, download and extract the Greengrass dependency checker from the GitHub repository, and then run the following commands:

```bash
git clone https://github.com/aws-samples/aws-greengrass-samples.git
cd aws-greengrass-samples
cd greengrass-dependency-checker-GGCv1.6.0
sudo ./check_ggc_dependencies
```

This script runs on AWS Greengrass supported platforms and requires the following Linux system commands:

```bash
printf, uname, cat, ls, head, find, zcat, awk, sed, sysctl, wc, cut, sort, expr, grep, test, dirname, readlink, xargs, strings, uniq
```

2. Install all required dependencies on your device, as indicated by the dependency script. 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.
Note
If no errors appear in the output, AWS Greengrass should be able to run successfully on your device.

For the list of AWS Greengrass requirements and dependencies, see Supported Platforms and Requirements (p. 12).

Configuring NVIDIA Jetson TX2 for AWS Greengrass

If your core device is an NVIDIA Jetson TX2, it must be configured before you can install the AWS Greengrass. The following steps describe how to flash the firmware to a JetPack installer and rebuild the kernel so that the device is ready to install the AWS Greengrass core software.

Note
The JetPack installer version that you use is based on your target CUDA Toolkit version. The following instructions assume that you're using JetPack 3.1 and CUDA Toolkit 8.0, because the binaries for TensorFlow v1.4.0 that AWS Greengrass provides for machine learning (ML) inference are compiled against this version of CUDA. For more information about AWS Greengrass ML inference, see Perform Machine Learning Inference (p. 154).

Flash the JetPack 3.1 Firmware

1. On a physical desktop that is running Ubuntu 14 or 16, flash the firmware to JetPack 3.1, as described in Download and Install JetPack L4T.

   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. Start the device in forced recovery mode.

   Note
   After the JetPack installation, you must use `ubuntu` credentials to log onto the device. The SSH agent hangs when it tries to log in using any other account, even if you SSH directly to the board using this account.

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

Rebuild the NVIDIA Jetson TX2 Kernel

Run the following commands on the Jetson board.

1. Check the kernel configurations:

   ```bash
   nvidia@tegra-ubuntu:~$ zcat /proc/config.gz | grep -e CONFIG_KEYS -
   + CONFIG_POSIX_MQUEUE -e CONFIG_OF_OVERLAY -e CONFIG_OVERLAY_FS -e
   + CONFIG_HAVE_ARCH_SECCOMP_FILTER -e CONFIG_SECCOMP -e CONFIG_SECCOMP
   -e CONFIG_DEVPTS_MULTIPLE_INSTANCES -e CONFIG_IPC_NS -e CONFIG_NET_NS -e
   CONFIG_UTS_NS -e CONFIG_USER_NS -e CONFIG_PID_NS -e CONFIG_CGROUPS -e CONFIG_MEMCG -e
   CONFIG_CGROUP_FREEZER -e CONFIG_CGROUP_DEVICE
   # CONFIG_POSIX_MQUEUE is not set
   CONFIG_CGROUPS=y
   CONFIG_CGROUP_FREEZER=y
   # CONFIG_CGROUP_DEVICE is not set
   # CONFIG_MEMCG is not set
   CONFIG_UTS_NS=y
   CONFIG_IPC_NS=y
   # CONFIG_USER_NS is not set
   CONFIG_PID_NS=y
   CONFIG_NET_NS=y
   CONFIG_HAVE_ARCH_SECCOMP_FILTER=y
   ```
2. Check the performance and power settings:

```bash
cuda@tegra-ubuntu:$ sudo nvpmodel -q
NV Power Mode: MAXP_CORE_ARM
```

3. Put the Jetson into high performance mode:

```bash
cuda@tegra-ubuntu:$ sudo nvpmodel -m 0
```

4. Clone the git repository:

```bash
cuda@tegra-ubuntu:$ cd /
cuda@tegra-ubuntu:$ sudo git clone https://github.com/jetsonhacks/buildJetsonTX2Kernel.git
```

5. Modify the `getKernelSources.sh` script, based on the following diff of the changes:

```bash
index f47f28d..3dd863a 100755
--- a/scripts/getKernelSources.sh
+++ b/scripts/getKernelSources.sh
@@ -1,12 +1,15 @@
 
 #!/bin/bash
 apt-add-repository universe
 apk-get update
-apt-get install qt5-default pkg-config -y
+apt-get install qt5-default pkg-config libncurses5-dev libssl-dev -y
 cd /usr/src
 wget http://developer.download.nvidia.com/embedded/L4T/r28_Release_v1.0/BSP/
 source_release.tbz2
 tar -xvf source_release.tbz2 sources/kernel_src-tx2.tbz2
 tar -xvf sources/kernel_src-tx2.tbz2
 cd kernel/kernel-4.4
 +make clean
 zcat /proc/config.gz > .config
 -make xconfig
 +echo "type something to continue"
 +read
 +make menuconfig
```

6. Run the `getKernelSources` script:

```bash
cuda@tegra-ubuntu:$ cd /buildJetsonTX2Kernel
cuda@tegra-ubuntu:$ sudo ./getKernelSources.sh
```

7. When prompted for "type something to continue", press `CTRL + Z` to background the script.

8. Go to `/usr/src/kernel/kernel-4.4/security/keys` and edit the `Kconfig` file by adding the following lines between `KEYS` and `PERSISTENT_KEYRINGS`:

```bash
config KEYS_COMPAT
def_bool y
depends on COMPAT & KEYS
```

9. Unpause the script:
Type some characters to unblock the script.

10. In the setup window that opens, choose **Enable loadable module support**, and then open the submenu to enable **optionModule** signature verification. Use the arrow keys to move and the spacebar to select any option. Then, save the change and exit.

11. Verify that **KEYS_COMPAT** is enabled:

```
$nvidia@tegra-ubuntu:~$ grep --color KEYS_COMPAT /usr/src/kernel/kernel-4.4/.config
```

12. Open the kernel configuration interface and enable kernel configurations:

```
$nvidia@tegra-ubuntu:~$ sudo make xconfig
```

A window opens that shows all the kernel configurations. Use **FIND** to search for the following keywords and tick-mark them.

**Note**

Keywords vary by configuration. The following list contains alternative versions in parentheses that can help you find the equivalent keywords for your configuration.

- **CONFIG_POSIX_MQUEUE** *(POSIX Message Queue)*
- **CONFIG_OF_OVERLAY** *(Overlay Filesystem Support)*
- **CONFIG_OVERLAY_FS** *(Overlay Filesystem Support)*
- **CONFIG_USER_NS** *(User Namespace)*
- **CONFIG_MEMCG** *(Memory Resource Controller for Control Group)*
- **CONFIG_CGROUP_DEVICE** *(Device Controller for cgroups)*

13. Build the kernel:

```
$nvidia@tegra-ubuntu:~$ cd /buildJetsonTX2Kernel
$nvidia@tegra-ubuntu:~$ sudo ./makeKernel.sh
```

14. Verify that the kernel configurations are enabled:

```
$nvidia@tegra-ubuntu:~$ grep --color CONFIG_POSIX_MQUEUE /usr/src/kernel/kernel-4.4/.config
$nvidia@tegra-ubuntu:~$ grep --color CONFIG_OF_OVERLAY /usr/src/kernel/kernel-4.4/.config
$nvidia@tegra-ubuntu:~$ grep --color CONFIG_OVERLAY_FS /usr/src/kernel/kernel-4.4/.config
$nvidia@tegra-ubuntu:~$ grep --color CONFIG_USER_NS /usr/src/kernel/kernel-4.4/.config
$nvidia@tegra-ubuntu:~$ grep --color CONFIG_MEMCG /usr/src/kernel/kernel-4.4/.config
$nvidia@tegra-ubuntu:~$ grep --color CONFIG_CGROUPDEVICE /usr/src/kernel/kernel-4.4/.config
$nvidia@tegra-ubuntu:~$ grep --color CONFIG_KEYS_COMPAT /usr/src/kernel/kernel-4.4/.config
$nvidia@tegra-ubuntu:~$ grep --color CONFIG_COMPAT /usr/src/kernel/kernel-4.4/.config
$nvidia@tegra-ubuntu:~$ grep --color CONFIG_KEYS /usr/src/kernel/kernel-4.4/.config
```

15. Copy the image:

```
$nvidia@tegra-ubuntu:~$ sudo ./copyImage.sh
```
Module 2: Installing the Greengrass Core Software

This module shows you how to install the AWS Greengrass core software on your device. Before you begin, make sure that you have completed Module 1 (p. 30).

The AWS Greengrass core software provides the following functionality:

- Allows deployment and execution of local applications that are created by using AWS Lambda functions and managed through the deployment API.
- Enables local messaging between devices over a secure network by using a managed subscription scheme through the MQTT protocol.
- Ensures secure connections between devices and the cloud using device authentication and authorization.
- Provides secure, over-the-air, software updates of user-defined Lambda functions.

This module should take less than 30 minutes to complete.

Configure AWS Greengrass on AWS IoT

1. Sign in to the AWS Management Console on your computer and open the AWS IoT console. If this is the first time opening this console, choose Get started.
2. Choose Greengrass.
4. Create an AWS Greengrass group. An AWS Greengrass group contains information about the devices and how messages are processed in the group. Each AWS Greengrass group requires an AWS Greengrass core device that processes messages sent within the group. An AWS Greengrass core needs a certificate and an AWS IoT policy to access AWS Greengrass and AWS Cloud Services. On the Set up your Greengrass group page, choose Use easy creation.

5. Type a name for your group (for example, MyFirstGroup), then choose Next.
6. Use the default name for the AWS Greengrass core, and choose Next.

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

8. On the confirmation page, download your core's security resources and the AWS Greengrass Core software, as follows:

   a. Under **Download and store your Core's security resources**, choose **Download these resources as a tar.gz** to download the required security resources for your AWS Greengrass core.

   ```
<table>
<thead>
<tr>
<th>Download and store your Core's security resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>A certificate for this Core: c6973960cc.cert.pem</td>
</tr>
<tr>
<td>A public key: c6973960cc.public.key</td>
</tr>
<tr>
<td>A private key: c6973960cc.private.key</td>
</tr>
<tr>
<td>Core-specific config file: config.json</td>
</tr>
</tbody>
</table>
   ```

   b. Under **Software configurations** for the AWS Greengrass core software, choose the CPU architecture and distribution (and operating system, if necessary) that best describes your core device:

   - If you're using a Raspberry Pi, download the ARMv7l for Raspbian Jessie package.
   - If you're using an Amazon EC2 instance, download one of the x86_64 packages.
You must download both the security resources and the AWS Greengrass Core software before you choose Finish.

9. After downloading the security resources and the AWS Greengrass Core software, choose Finish.

The group configuration page is displayed in the console:

Start AWS Greengrass on the Core Device

1. In the prior step (p. 47), you downloaded two files from the AWS Greengrass console:

   - `greengrass-OS-architecture-1.6.0.tar.gz` - this compressed file contains the AWS Greengrass Core software that runs on the AWS Greengrass core device.
   - `GUID-setup.tar.gz` - this compressed file contains security certificates enabling secure communications with the AWS IoT cloud and config.json which contains configuration information specific to your AWS Greengrass core and the AWS IoT endpoint.

If you don’t recall the IP address of your AWS Greengrass core device, open a terminal on the AWS Greengrass core device and run the following command:
hostname -I

Based on your operating system, choose a tab to transfer the two compressed files from your computer to the AWS Greengrass core device:

**Note**
Recall that the default login and password for the Raspberry Pi is **pi** and **raspberry**, respectively.

**Windows**

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

```bash
cd path-to-downloaded-files
pscp -pw Pi-password greengrass-OS-architecture-1.6.0.tar.gz pi@IP-address:/home/pi
pscp -pw Pi-password GUID-setup.tar.gz pi@IP-address:/home/pi
```

For example:

```
C:\\cd Users\\Downloads
C:\Users\\Downloads\\pscp -pw raspberries@851sf2090-set us.tar.gz pi@10.0.0.12:/home/pi
851sf2090-set us.tar.gz | 2 KB | 2.7 KB/s | ETA: 00:00:01 | 100%
C:\Users\\Downloads\\pscp -pw raspberries@851sf2090-set us.tar.gz pi@10.0.0.12:/home/pi
```

**macOS**

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

**Note**
You may be prompted for two passwords. If so, the first password is for the Mac's `sudo` command and the second will be the password for the Raspberry Pi.

```bash
cd path-to-downloaded-files
sudo scp greengrass-OS-architecture-1.6.0.tar.gz pi@IP-address:/home/pi
sudo scp GUID-setup.tar.gz pi@IP-address:/home/pi
```

**UNIX-like system**

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

```bash
cd path-to-downloaded-files
sudo scp greengrass-OS-architecture-1.6.0.tar.gz pi@IP-address:/home/pi
sudo scp GUID-setup.tar.gz pi@IP-address:/home/pi
```

**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 (i.e., `~/home/pi/Downloads`). Otherwise, the compressed files should be in the Pi's `~` folder (i.e., `/home/pi`).
Open a terminal on the AWS Greengrass core device and navigate to the folder containing the compressed files (i.e., `path-to-compressed-files`).

```
cd path-to-compressed-files
```

Next, run the following commands to decompress the AWS Greengrass core binary file and the security resources (certificates, etc.) file:

```
sudo tar -xzvf greengrass-OS-architecture-1.6.0.tar.gz -C /
sudo tar -xzvf GUID-setup.tar.gz -C /greengrass
```

Among other things, the first command creates the `/greengrass` directory in the root folder of the AWS Greengrass core device (via the `-C /` argument). The second command copies the certificates into the `/greengrass/certs` folder and the `config.json` file into the `/greengrass/config` folder (via the `-C /greengrass` argument). For more information, see AWS Greengrass Core Configuration File (p. 17).

2. Install the Symantec VeriSign root CA onto your device. This certificate enables your device to communicate with AWS IoT using the MQTT messaging protocol over TLS. Make sure the AWS Greengrass core device is connected to the internet, then run the following commands (note that `-O` is the capital letter O):

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

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.

3. Use the following commands to start AWS Greengrass.

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

You should see output similar to the following (note the PID number):

```
pi@raspberrypi:/greengrass/ggc/core $ sudo ./greengrassd start
Setting up greengrass daemon
Validating hardlink/softlink protection
Validating execution environment
Found cgroup subsystem: cpu
Found cgroup subsystem: cpucfs
Found cgroup subsystem: blkio
Found cgroup subsystem: memory
Found cgroup subsystem: devices
Found cgroup subsystem: freezer
Found cgroup subsystem: net_cls
Starting greengrass daemon
Greengrass successfully started with PID: 2244
```
Module 3 (Part 1): Lambda Functions on AWS Greengrass

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

Part 1 of this module shows you how to deploy a Lambda function on the AWS Greengrass core that sends "Hello World" messages to the AWS Greengrass cloud. Part 2 (p. 66) covers the differences between on-demand and long-lived Lambda functions running on the AWS Greengrass core. Before you begin, make sure that you have completed Module 1 (p. 30) and Module 2 (p. 43) and have a running AWS Greengrass core device. Note that Module 3 (Part 1) (p. 51) and Module 3 (Part 2) (p. 66) should take approximately 30 minutes each.

Create and Package a Lambda Function

In order for a Python Lambda function to run on an AWS Greengrass core, it must be packaged with specific folders from the Python AWS Greengrass Core SDK. In the following, you will:

- Download the Python AWS Greengrass Core SDK to your computer (not the AWS Greengrass core device).
- Decompress the downloaded SDK file.
- Obtain the Python Lambda function (named `greengrassHelloWorld.py`) from the decompressed SDK.
- Create a Lambda function deployment package named `hello_world_python_lambda.zip` that contains `greengrassHelloWorld.py` and the `greengrasssdk` folder.
- Upload the `hello_world_python_lambda.zip` package by using the Lambda console.
- Transfer the package to the AWS Greengrass core device by using the AWS Greengrass console.

1. In the AWS IoT console, choose **Software**.
2. Under **SDKs**, for **AWS Greengrass Core SDK**, choose **Configure download**.

3. Choose **Python 2.7 version 1.2.0**, and then choose **Download Greengrass Core SDK**.
4. Decompress the downloaded SDK. For instructions, choose the tab that corresponds to your operating system.

Windows

Use a tool for decompressing .tar.gz files on Windows such as 7-Zip, WinZip, or similar. As an example, the 7-Zip tool can be used to decompress greengrass-core-python-sdk-1.2.0.tar.gz as follows:

1. After installing 7-Zip, navigate to the greengrass-core-python-sdk-1.2.0.tar.gz file using Windows File Explorer (Windows logo key + E), right-click the file, choose 7-Zip, then choose Open archive.
2. In the resulting 7-Zip window, double-click greengrass-core-python-sdk-1.2.0.tar, aws_greengrass_core_sdk, examples, HelloWorld, and then greengrassHelloWorld.zip.
3. Optionally using the Ctrl key, select the greengrasssdk folder and the Python greengrassHelloWorld.py Lambda file. Next, choose Extract, pick a location to extract the files to, and choose OK.

macOS

1. Using Finder, navigate to the greengrass-core-python-sdk-1.2.0.tar.gz file and double-click it. This creates the aws_greengrass_core_sdk folder.
2. Expand the aws_greengrass_core_sdk folder, then the examples folder, and then the HelloWorld folder.
3. Double-click the greengrassHelloWorld.zip file. This creates the greengrassHelloWorld folder – expand this folder.

UNIX-like system

1. Open a terminal window and navigate to the directory containing the greengrass-core-python-sdk-1.2.0.tar.gz file.
2. Run the following command to decompress the file:

```
sudo tar -xzf greengrass-core-python-sdk-1.2.0.tar.gz
```

This creates the aws_greengrass_core_sdk directory. Next, run the following commands:

```
cd /aws_greengrass_core_sdk/examples/HelloWorld
sudo unzip greengrassHelloWorld.zip
```

You use the greengrasssdk folder and the Python greengrassHelloWorld.py Lambda function code in the next step.
Note that every five seconds, the `greengrassHelloWorld.py` Lambda function publishes one of two possible messages to the `hello/world` topic, as shown in the following code (to save space, all code comments have been removed):

```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
```

5. To create the `greengrassHelloWorld.py` Lambda function, you must package the `greengrassHelloWorld.py` file and the `greengrasssdk` folder into a compressed `.zip` file. This is your Lambda function deployment package. For this tutorial, name the package `hello_world_python_lambda.zip`:

For UNIX-like systems (including the Mac terminal), this can be accomplished with the following command:

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

Note
---
Depending on your distribution, you may need to install `zip` first. For example, `sudo apt-get install zip` (this installation command may differ for your distribution).

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

6. Open the Lambda console and choose **Create function**.

7. Choose **Author from scratch**.
8. Name your function **Greengrass_HelloWorld**, and set the remaining fields as follows:
   - **Runtime** - choose **Python 2.7**.
   - **Role** - choose **Create new role from templates(s)**.
   - **Role name** - type a unique name for the role.

   Note that this role isn't used by AWS Greengrass.

   Then, choose **Create function**.

9. Upload your Lambda function deployment package, as follows:
   a. On the **Configuration** tab, under **Function code**, set the following fields:
      - **Code entry type** - choose **Upload a .ZIP file**.
      - **Runtime** - choose **Python 2.7**.
      - **Handler** - type **greengrassHelloWorld.function_handler**.
b. Choose **Upload**, and then choose `hello_world_python_lambda.zip`. Your `hello_world_python_lambda.zip` file size may vary.

c. Choose **Save**.

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

10. Publish the Lambda function, as follows:

a. From the **Actions** menu, choose **Publish new version**:

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

11. Create an **alias** for the Lambda function version, as follows:

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

---

**Configure the Lambda Function for AWS Greengrass**

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

1. In the AWS IoT console, under **Greengrass**, choose **Groups**, and then choose the group that you created in Module 2 (p. 43).
2. On the group configuration page, choose **Lambdas**, and then choose **Add Lambda**.
3. **Choose Use existing Lambda.**

   **Add a Lambda to your Greengrass Group**

   Local Lambdas are hosted on your Greengrass Core and connected to each other and devices by Subscriptions, but they can also be deployed individually to your Group.

   **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.

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 (…) for the Lambda function, then choose Edit Configuration:**
AWS Greengrass Developer Guide  
Configure the Lambda Function for AWS Greengrass

7. On the Group-specific Lambda configuration page, make the following changes:
   - For **Timeout** - set to 25 seconds. This Lambda function sleeps for 20 seconds before each invocation.
   - For **Lambda lifecycle** - select **Make this function long-lived and keep it running indefinitely**.

A *long-lived* Lambda function starts automatically after AWS 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. 125).

**Note**
The Lambda function in this tutorial accepts **JSON** input payloads, but Greengrass Lambda functions also support **Binary** input payloads. Binary support is especially useful when interacting with device data, because the restricted hardware capabilities of devices often make it difficult or impossible to construct a JSON data type.

8. Choose **Update** to save your changes to the Lambda function configuration.

9. An AWS Greengrass Lambda function can subscribe or publish messages (using the **MQTT protocol**):
   - To and from other devices (or device shadows) within the AWS Greengrass core. Information about device shadows is provided in **Module 5** (p. 91).
   - To other Lambda functions.
   - To the AWS IoT cloud.
The AWS Greengrass group controls the way in which these components interact by using subscriptions that enable greater security and to provide predictable interactions.

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 an AWS Greengrass device, a Lambda function, a device shadow, or the AWS IoT cloud. A subscription is directed in the sense that messages flow in a specific direction. For an AWS Greengrass device to send messages to and receive messages from a Lambda function, you must set up two subscriptions: one from the device to the Lambda and another from the Lambda function to the device. The Greengrass_HelloWorld Lambda function sends messages only to the hello/world topic in the AWS IoT cloud, 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))

    # Asynchronously schedule this function to be run again in 5 seconds
    Timer(5, greengrass_hello_world_run).start()

# Execute the function above:
greengrass_hello_world_run()
```

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

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

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

13. In the **Optional topic filter** field, type `hello/world`, then choose **Next**.
14. Finally, choose Finish.

Deploy Cloud Configurations to an AWS Greengrass Core Device

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

2. Make sure that the AWS 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:
      ```bash
      ps aux | grep -E 'greengrass.*daemon'
      ```
      If the output contains a root entry for /greengrass/ggc/packages/1.6.0/bin/daemon, then the daemon is running.
   b. To start the daemon:
      ```bash
      cd /greengrass/ggc/core/
      sudo ./greengrassd start
      ```

Now you're ready to deploy the Lambda function and subscription configurations to your AWS Greengrass core device.

3. In the AWS IoT console, on the group configuration page, from the Actions menu, 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 Greengrass also supports manually specified endpoints. You're only prompted for the discovery method the first time that the group is deployed.
5. If prompted, choose Grant permission on the Grant permission to access other services page. This creates the Greengrass service role, which allows AWS Greengrass to access other AWS services on your behalf. This role is required for deployments to succeed. You need to do this only once per account.

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:

For help troubleshooting any issues that you encounter, see Troubleshooting AWS Greengrass Applications (p. 196).

Verify the Lambda Function Is Running on the Device

From the left pane of the AWS IoT console, choose Test.
In Subscription topic field, type hello/world (don't choose Subscribe to topic yet). For Quality of Service, select 0. For MQTT payload display, select Display payloads as strings (more accurate).
Next, choose **Subscribe to topic**.

Assuming the Lambda function is running on your device, it will publish messages to the **hello/world** topic similar to the following:
Module 3 (Part 2): Lambda Functions on AWS Greengrass

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

Part 1 (p. 51) of this module described how to deploy a Lambda function on a AWS Greengrass core that sends "Hello World" messages to the AWS Greengrass cloud. This part explores the differences between on-demand and long-lived Lambda functions running on the AWS Greengrass core. Before you begin, make sure you have completed Module 1 (p. 30), Module 2 (p. 43), and Module 3 (Part 1) (p. 51). This module should take approximately 30 minutes to complete.

Create and Package the Lambda Function

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

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

2. Package the greengrassHelloWorldCounter.py file with the SDK into a .zip file, as described in Module 3 (Part 1) (p. 51). 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. 51). You can use the existing role.
4. Upload your Lambda function deployment package, as follows:
   a. On the **Configuration** tab, under **Function code**, set the following fields:
      - **Code entry type** - choose **Upload a .ZIP file**.
      - **Runtime** - choose **Python 2.7**.
      - **Handler** - type **greengrassHelloWorldCounter.function_handler**.
   b. Choose **Upload**, and then choose **hello_world_counter_python_lambda.zip**.
   c. At the top of the page, choose **Save**.

5. Publish the first version of the function, as follows:
   a. From the **Actions** menu, choose **Publish new version**. For **Version description**, type **First version**.
   b. Choose **Publish**.

6. Create an alias for the function version, as follows:
   a. From the **Actions** menu, choose **Create alias**, and set the following values:
      - **Name** - type **GG_HW_Counter**.
      - **Version** - choose **1**.
   b. Choose **Create**.

Recall that aliases create a single entity for your Lambda function that AWS Greengrass devices can subscribe to without having to update subscriptions with Lambda version numbers every time the function is modified.
Configure Long-Lived Lambda Functions for AWS Greengrass

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

1. In the AWS IoT console, under Greengrass, choose Groups, and then choose the group that you created in Module 2 (p. 43).
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.

   ![Use existing Lambda page](image)

5. On the Select a Lambda version page, choose Alias: GG_HW_Counter, and then choose Finish.
6. On the Lambdas page, from the Greengrass_HelloWorld_Counter menu, choose Edit Configuration.

   ![Edit Configuration](image)

7. On the configuration page, edit the following properties:
   - **Timeout** - set to 25 seconds. This Lambda function sleeps for 20 seconds before each invocation.
   - **Lambda lifecycle** - choose Make this function long-lived and keep it running indefinitely.
Test Long-Lived Lambda Functions

A *long-lived Lambda function* starts automatically when the AWS 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 `greengrassHelloWorldCounter.py` Lambda function is similar to the `greengrassHelloWorld.py` function except there is a variable, `my_counter`, that is outside of the `function_handler(event, context)` method (code comments were removed for brevity):

```python
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

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

   ![Select your source and target](image)

   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)

   For Optional topic filter, type hello/world/counter. Choose Next and then choose Finish.

   ![Optional topic filter](image)

   **Source**
   - Greengrass_HelloWorld_Counter (LAMBDA)

   **Optional topic filter**
   - hello/world/counter

   **Target**
   - IoT Cloud (SERVICE)
This single subscription goes in one direction only: from the `Greengrass_HelloWorld.Counter` Lambda function to the AWS IoT cloud. To trigger this Lambda function from the cloud, you need to create a subscription in the opposite direction.

2. Add another subscription with **IoT Cloud** as the source and **Greengrass_HelloWorld.Counter** as the target. Use the `hello/world/counter/trigger` topic:

```
CREATE A SUBSCRIPTION
Confirm and save your Subscription
```

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.

```
IoT Cloud
Service
hello/world/counter/trigger
Greengrass_HelloWorld.Counter
```

Note the `/trigger` extension above – because you have created two subscriptions, you do not want them to interfere with each other.

3. Make sure that the AWS Greengrass daemon is running, as described in **Deploy Cloud Configurations to a Core Device** (p. 62).

Note that with the daemon running, the prior `greengrassHelloWorld.py` Lambda function will continue to send messages to the `hello/world` topic (in the AWS IoT cloud). This does not, however, interfere with the messages sent from the `greengrassHelloWorldCounter.py` Lambda function to the AWS IoT cloud, since they're directed to a different topic, namely `hello/world/counter`.

4. On the group configuration page, from the **Actions** menu, choose **Deploy** to deploy the updated group configuration to your AWS Greengrass core device.

```
MyFirstGroup
Successfully completed
```

For help troubleshooting any issues that you encounter, see **Troubleshooting AWS Greengrass Applications** (p. 196).

5. After your deployment is complete, in the AWS IoT console, choose **Test**. In **Subscription topic**, type `hello/world/counter`. For **Quality of Service**, select 0. For **MQTT payload display**, select **Display payloads as strings**, and then choose **Subscribe to topic**.
Unlike Part 1 (p. 51) of this module, you should *not* be able to see any messages after you subscribe to **hello/world/counter**. This is because the `greengrassHelloWorldCounter.py` code to publish to the topic **hello/world/counter** is inside the `function_handler(event, context)` function, and `function_handler(event, context)` is triggered only when it receives an MQTT message on the **hello/world/counter/trigger** topic. To help further explain this, consider the **greengrass_HelloWorld_Counter** related subscriptions:

In the second row, we see that the **greengrass_HelloWorld_Counter** Lambda function can send messages to the **IoT Cloud** on the **hello/world/counter** topic. In the third row, we see that the **IoT Cloud** will can send messages to the **greengrass_HelloWorld_Counter** Lambda function when that message is sent to the **hello/world/counter/trigger** topic (note that there is nothing special about the word `trigger`). The **greengrass_HelloWorld_Counter** Lambda function ignores these sent messages and merely runs the code within `function_handler(event, context)`, which sends a message back to the **hello/world/counter** topic in the AWS IoT cloud (see the prior `greengrassHelloWorldCounter.py` code listing).
So, to trigger the `function_handler(event, context)` handler, publish any message (the default message is fine) to the `hello/world/counter/trigger` topic, as shown next.

Every time a message is published to the `hello/world/counter/trigger` topic, the `my_counter` variable is incremented (see Invocation Count in the following). Because the function handler in the Lambda function includes a 20-second sleep cycle (i.e., `time.sleep(20)`), repeatedly triggering the handler queues up responses from the AWS Greengrass core.
Test On-Demand Lambda Functions

An on-demand Lambda function is similar in functionality to an AWS cloud 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 new containers are created. For more information, see the section called “Lifecycle Configuration” (p. 125).

1. On the group configuration page, choose Lambdas. For the Greengrass_HelloWorld_Counter Lambda function, choose Edit Configuration.

2. Under Lambda lifecycle, select On-demand function.

   Alias GG_HW_Counter

   Memory limit

   16 MB

   Timeout

   25 Sec...

   Lambda lifecycle

   On-demand function

   Make this function long-lived and keep it running indefinitely

   Next, choose Update.

3. On the group configuration page, from the Actions menu, choose Deploy to deploy the updated group configuration to your AWS Greengrass core device.
4. After your deployment is complete, in the AWS IoT console, choose Test. For **Subscription topic**, type `hello/world/counter`. For **Quality of Service**, select 0. For **MQTT payload display**, select **Display payloads as strings** and then choose **Subscribe to topic**.

Again, you should not be able to see any messages after you subscribe. Trigger the function to the `hello/world/counter/trigger` topic by sending any message (the default message is fine), then choose **Publish to topic** three times, **within five seconds** of each press of the button.
Each publish is triggering the function handler and creating a new container for each invocation. The invocation count is not incremented for each of the three times you triggered the function because each on-demand Lambda function has its own container/sandbox.

Wait approximately thirty seconds or more, and then choose Publish to topic. This time you should see an incremented invocation count.
This shows that a container, first created from a prior invocation, is being reused, and preprocessing variables outside of the function handler have been stored.

You should now understand the two types of Lambda functions that can run on the AWS Greengrass core. The next module, Module 4 (p. 77), shows you how devices can interact within an AWS Greengrass group.

Module 4: Interacting with Devices in an AWS Greengrass Group

This module shows you how AWS IoT devices can connect to and communicate with an AWS Greengrass core device. AWS IoT devices that connect to an AWS Greengrass core are part of an AWS Greengrass group and can participate in the AWS Greengrass programming paradigm. In this module, one AWS Greengrass device sends a "Hello World" message to another AWS Greengrass device within the AWS Greengrass group:
Before you begin, make sure that you have completed Module 1 (p. 30), Module 2 (p. 43), Module 3 (Part 1) (p. 51), and Module 3 (Part 2) (p. 66). You do not need other components or devices. This module should take less than 30 minutes to complete.

**Create AWS IoT Devices in an AWS Greengrass Group**

1. In the AWS IoT console, choose **Greengrass**, choose **Groups**, and then choose your group to open its configuration page. Next, choose **Devices** and then choose **Add your first Device**.
Choose Create New Device:

Add a Device

Greengrass Devices can be created by re-purposing an existing IoT Thing from your Registry or by creating new Registry items, and then adding them to a Greengrass Group.

Create a new Device
You will create a new Device and generate a certificate, a private key and a public key.

Use an existing IoT Thing as an Device
You can add an existing IoT Thing to your Group.

Register this device as HelloWorld_Publisher, then choose Next:
For **1-Click**, choose **Use Defaults**: 

The Device needs a certificate and a policy before it can be added to Greengrass. If you’re unfamiliar with any of these steps we recommend the Automated Setup.

**1-Click**

This will generate a certificate, public key and private key using AWS IoTs root CA, generate a default policy, and create a new IAM role with default permissions.

**Advanced setup**

This will allow you to handle certificate signing request (CSR) based on a private key you own, customize your own policy, and use an existing IAM role or create a new one.

Create a folder on your computer.

Download the certificates for your device into the folder, and then decompress them (to decompress tar.gz files on Windows, see the **Windows** tab in step 2 of Create and Package a Lambda Function (p. 51)).
Create AWS IoT Devices in an AWS Greengrass Group

Note the common GUID-like filename component for the `HelloWorld_Publisher` device (in this example, 51d2737e90), this will be needed later. Finally, choose **Finish**.

2. By choosing **Add Device**, repeat step 1 to add another device to the group and name it `HelloWorld_Subscriber`. Download the certificates for your second device onto your computer as well, saving them in the same folder as the first set of certificates. Choose **Finish**. Again, note the common GUID-like filename component for the `HelloWorld_Subscriber` device.

You should now have two devices in your AWS Greengrass group:

3. Download another **AWS IoT root certificate from Symantec** and save it as `root-ca-cert.pem` in the folder you just created. For this module, this certificate and the certificates and keys for both devices should be in one folder on your computer (not on the AWS Greengrass core device).
Note
If you’re using a web browser on the Mac and you receive a This certificate is already
installed as a certificate authority alert, you can open a Terminal window and run
the following commands to download the certificate into the folder containing the
HelloWorld_Publisher and HelloWorld_Subscriber device certificates/keys:

cd path-to-folder-containing-device-certificates
curl -o ./root-ca-cert.pem http://www.symantec.com/content/en/us/enterprise/
verisign/roots/VeriSign-Class%203-Public-Primary-Certification-Authority-G5.pem

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

Configure Subscriptions

In this step, you enable the HelloWorld_Publisher device to send a HelloWorld message to the
HelloWorld_Subscriber device.

1. On the group configuration page, choose Subscriptions, and then choose Add Subscription. For
Select a source, choose Select, Devices, and HelloWorld_Publisher. Similarly, for Select a target,
choose HelloWorld_Subscriber. Lastly, choose Next:

For Optional topic filter, type hello/world/pubsub:

Then choose Next followed by Finish.
Install the AWS IoT Device SDK for Python

The AWS IoT Device SDK for Python can be used by all AWS IoT devices to communicate with the AWS IoT cloud and AWS Greengrass core devices (using the Python programming language). Note that the SDK requires Python 2, version 2.7+ or Python 3, version 3.3+. Additionally, the SDK requires OpenSSL version 1.0.1+ (TLS version 1.2) compiled with the Python executable.

To install the SDK onto your computer, with all required components, choose the appropriate tab:

Windows
1. Open an elevated command prompt and 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, then install Python 2.7+ or Python 3.3+ by following the instructions in Downloading Python. For additional information, see Using Python on Windows (note the Python version number drop-down menu).

2. Using a web browser, download the AWS IoT Device SDK for Python zip file and and save it as aws-iot-device-sdk-python-latest.zip (this should be the default name). The zip file will typically be saved to your Downloads folder. Decompress aws-iot-device-sdk-python-latest.zip to an appropriate location, such as your home directory (i.e., cd %HOME %). Note the file path to the decompressed aws-iot-device-sdk-python-latest folder. In the next step, this file path will be indicated by path-to-SDK-folder.

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

   cd path-to-SDK-folder

Note
You may delete subscriptions from the earlier modules. To do so, choose Subscriptions, choose an ellipsis (…) associated with a subscription, and then choose Delete.

2. Make sure that the AWS Greengrass daemon is running, as described in Deploy Cloud Configurations to a Core Device (p. 62).

3. On the group configuration page, from the Actions menu, choose Deploy to deploy the updated group configuration to your AWS Greengrass core device:

   To confirm a successful deployment, choose Deployments and you should see a Successfully completed message near the time you initiated the deployment.

   For help troubleshooting any issues that you encounter, see Troubleshooting AWS Greengrass Applications (p. 196).
Install the AWS IoT Device SDK for Python

```
python setup.py install
```

**macOS**

1. Open a Terminal window and 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, then install Python 2.7+ or Python 3.3+ by following the instructions in Downloading Python. For additional information, see Using Python on a Macintosh (note the Python version number drop-down menu).

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

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

Note the OpenSSL version value.

**Note**

Use `print(ssl.OPENSSL_VERSION)` if you’re running Python 3.

To close the Python shell, run the following command:

```
>>> exit()
```

If the OpenSSL version is 1.0.1 or later, skip to step 3. Otherwise, proceed as follows:

- From the Terminal window, run the following command to determine if the computer is using Simple Python Version Management:

```
which pyenv
```

If a file path is returned, then choose the **Using pyenv** tab. If nothing is returned, choose the **Not using pyenv** tab.

**Using pyenv**

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

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

```
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 would be:

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

3. Close the Terminal window, then reopen it.

4. In the reopened Terminal window, run the following commands:
Install the AWS IoT Device SDK for 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 prior `pyenv install` and `pyenv global` commands and try again.

5. 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 and be aware that 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
```

Recall that the AWS IoT Device SDK for Python requires OpenSSL version 1.0.1 (or later) compiled with the Python executable. The prior `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 this, run the following command:

```bash
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:

```bash
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:

```bash
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
```

Now, invoking the python command will run the Python executable containing the required OpenSSL version (i.e., 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()
```

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
```

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 that 2.7 for Python 2 or less than 3.3 for Python 3, then install Python 2.7+ or Python 3.3+ by following the instructions in Downloading Python. For additional information, see Using Python on Unix platforms (note the Python version number drop-down menu)

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

1. Make sure that your computer and the AWS Greengrass core device are connected to the internet and are using the same network. The following procedure can be used to confirm this:
   a. Determine the IP address of the AWS Greengrass core by running the following command:

```
hostname -I
```
b. From your computer, run the following command using the core's IP address (you can use Ctrl + C to halt the ping command):

```
ping IP-address
```

Output similar to the following indicates successful communication between the computer and the AWS Greengrass core device (note 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
```

2. In the AWS IoT console, choose **Settings**.
3. Under **Settings**, note the value of **Endpoint**. In the following examples, this endpoint value is indicated by `AWS_IOT_ENDPOINT`.

**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**

```
abcdefg1j1234.iot.us-west-2.amazonaws.com
```
4. Choose **Greengrass, Groups**, and then choose your group. Choose **Settings**, select **Manually manage connection information**, then choose **View Cores for specific endpoint information**. Choose your core, then **Connectivity**. Choose **Edit** and ensure you have only one **Endpoint** value which must be the IP address of your AWS Greengrass core device (for port 8883). Choose **Update**. This ensures that the `basicDiscovery.py` script will connect to the correct AWS Greengrass core device IP address.

5. Open two command-line windows on your computer (not the AWS Greengrass core device). One command-line window will be for the **HelloWorld_Publisher** device and the other for the **HelloWorld_Subscriber** device. Every time the following script (`basicDiscovery.py`) is executed for the first time, the AWS Greengrass discovery service connects to the AWS Greengrass core. After a device has discovered the AWS Greengrass core and successfully connected to it, future messaging and operations can be executed locally (without the need for an internet connection).

**Note**
You can run the following command from the folder containing the `basicDiscovery.py` file for detailed script usage information:

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

From the **HelloWorld_Publisher** device window, run the following commands:

```
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:

```
Published topic hello/world/pubsub: {'message': "Hello, World! Sent from HelloWorld_Publisher", "sequence": 0}
2017-11-13 21:12:27,301 - AWSIoTPythonSDK.core.protocol.mqtt_core - INFO - Performing sync publish...
Published topic hello/world/pubsub: {'message': "Hello, World! Sent from HelloWorld_Publisher", "sequence": 1}
2017-11-13 21:12:28,306 - AWSIoTPythonSDK.core.protocol.mqtt_core - INFO - Performing sync publish...
Published topic hello/world/pubsub: {'message': "Hello, World! Sent from HelloWorld_Publisher", "sequence": 2}
2017-11-13 21:12:28,387 - AWSIoTPythonSDK.core.protocol.mqtt_core - INFO - Performing sync publish...
Published topic hello/world/pubsub: {'message': "Hello, World! Sent from HelloWorld_Publisher", "sequence": 3}
```

**Note**
If you get a message about unrecognized arguments, try changing the single quotes to double quotes for the message parameter.

6. From the **HelloWorld_Subscriber** device window, run the following commands:

```
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:
Notice that closing the HelloWorld_Publisher window halts additional messages from accruing in the HelloWorld_Subscriber window.

For information about the latest usage of basicDiscovery.py, see the README file of the AWS IoT Device SDK for Python.

Module 5: Interacting with Device Shadows

This advanced module shows you how AWS Greengrass devices can interact with AWS IoT device shadows in an AWS 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 Greengrass device (GG_Switch) can modify the state of another AWS Greengrass device (GG_TrafficLight) and how these states can be synced to the AWS Greengrass cloud:

![Diagram of interaction between devices]
Before you begin, make sure that you have completed Module 1 (p. 30), Module 2 (p. 43), Module 3 (Part 1) (p. 51), and Module 3 (Part 2) (p. 66). You should also understand how to connect devices to an AWS Greengrass core (Module 4 (p. 77)). You do not need other components or devices. This module should take about 30 minutes to complete.

Configure Devices and Subscriptions

1. Create two devices in your AWS Greengrass group, `GG_Switch` and `GG_TrafficLight`. Use the default security settings.

   ![Device List Screen]

   Save the certificates for the devices to your computer – note the GUID-like filename component for the `GG_Switch` and `GG_TrafficLight` devices, these will be needed later. You can reuse the previous root CA from VeriSign or download a new one.

   Now, each shadow can be synced to AWS IoT when the AWS Greengrass core is connected to the internet. First, you'll use local shadows without syncing the shadows to the cloud. Later in the module, you enable syncing. By default, cloud syncing should be disabled. If it's not disabled, under Devices, choose the ellipsis (...), and then choose Local Shadow Only.

2. Choose Subscriptions and create the following subscriptions for your group (For information on the $ sign, see Reserved Topics). For example, to set up the first row subscription, choose Add Subscription, for Select a source choose Select, choose the Devices tab, and then choose `GG_Switch`. For Select a target choose Select, choose Local Shadow Service, and then Next. For Optional topic filter, type (or copy/paste) `$aws/things/GG_TrafficLight/shadow/update`, choose Next, and then Finish. Using a similar procedure, complete the remaining subscriptions:
### Configure Devices and Subscriptions

<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>$aws/things/GG_TrafficLight/shadow/update</td>
<td>The GG_Switch sends an update request to update topic.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$aws/things/GG_TrafficLight/shadow/update/accepted</td>
<td>The GG_Switch needs to know whether the update request was accepted or rejected.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$aws/things/GG_TrafficLight/shadow/update/rejected</td>
<td>The GG_Switch needs to know whether the update request was accepted or rejected.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$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></td>
<td>GG_TrafficLight</td>
<td>$aws/things/GG_TrafficLight/shadow/update/accepted</td>
<td>The GG_TrafficLight needs to know whether the update request from GG_Switch was accepted or rejected.</td>
</tr>
<tr>
<td></td>
<td>GG_TrafficLight</td>
<td>$aws/things/GG_TrafficLight/shadow/update/rejected</td>
<td>The GG_TrafficLight needs to know whether the update request from GG_Switch was accepted or rejected.</td>
</tr>
</tbody>
</table>

**Note**
Although you can use wildcards (for example, $aws/things/GG_TrafficLight/shadow/#) to consolidate some of the subscriptions, we do not recommend this practice.

The topic paths must be written exactly as shown in the table. Do not include an extra / at the end of a topic. You can hover your mouse over a Topic path to see the full path via tooltip popup:
Download Required Files

1. If you haven’t already done so, install the AWS IoT Device SDK for Python. Follow the instructions in the README file. This SDK is used by all AWS IoT devices to communicate with the AWS IoT cloud and AWS Greengrass cores.

2. From the AWS Greengrass samples repository on GitHub, download the `lightController.py` and `trafficLight.py` files to your computer and move them to the folder containing the GG_Switch and GG_TrafficLight device certificates.

---

**Note**

Each device has its own device shadow service. For more information, see Shadow MQTT Topics.

3. Make sure that the AWS Greengrass daemon is running, as described in Deploy Cloud Configurations to a Core Device (p. 62).

4. On the group configuration page, from the Actions menu, choose Deploy to deploy the updated group configuration to your AWS Greengrass core device.

For help troubleshooting any issues that you encounter, see Troubleshooting AWS Greengrass Applications (p. 196).
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. Open two command-line windows on your computer (not the AWS Greengrass core device). One command-line window is for the GG_Switch device and the other is for the GG_TrafficLight device. Both scripts, when executed for the first time, will run the AWS Greengrass discovery service to connect to the AWS Greengrass core (through the internet). After a device has discovered and successfully connected to the AWS Greengrass core, future operations can be executed locally. Before running the following commands, make sure that your computer and the AWS Greengrass core are connected to the internet using the same network.

For the GG_Switch command-line window, run the following:

```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
```

For the GG_TrafficLight command-line window, run the following:

```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
```

**Note**

To find the endpoint to use for the `AWS_IOT_ENDPOINT` placeholder value, open the AWS IoT console and choose **Settings**.

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:**
**Test Communications (Device Syncs Enabled)**

1. In the AWS IoT console, choose your AWS Greengrass group, **Devices**, and then **GG_TrafficLight**:

![AWS Greengrass Group](image)

Choose **Shadow**. For **Shadow State**, there should not be any updates to this shadow topic after the **GG_Switch** changes states because the **GG_TrafficLight** is set to **LOCAL SHADOW ONLY** as opposed to **SHADOW SYNCRECNG TO CLOUD**, as discussed in the next section.

2. Press Ctrl + C in the **GG_Switch** (lightController.py) command-line window and note that the **GG_TrafficLight** (trafficLight.py) window stops receiving state change messages.

3. In the AWS IoT console, choose your AWS Greengrass group, choose **Devices**, choose the ellipsis for the **GG_TrafficLight** device, then choose **Sync to the Cloud**:
2. From the **Deployments** page, deploy the updated configuration to your AWS Greengrass core device.

3. Repeat the step in which you create two command-line windows (p. 95).

4. In the AWS IoT console, choose your AWS Greengrass group, **Devices**, **GG_TrafficLight**, and then **Shadow**.

   Because you enabled syncs of the **GG_TrafficLight** shadow to AWS IoT, the shadow state in the cloud should be updated automatically whenever **GG_Switch** sends an update. This functionality can be used to expose the state of an AWS Greengrass device to the AWS IoT cloud.

   **Shadow Document**

   **Last update:** Jan 9, 2018 3:39:53 PM -0800

   **Shadow state:**

   ```json
   {
   "desired": {
       "property": "G"
   },
   "reported": {
       "property": "G"
   }
   }```
Module 6: Accessing AWS Cloud Services

This advanced module shows you how AWS Greengrass cores can interact with other Amazon Web Services in the cloud. It builds on the traffic light example in Module 5 (p. 91) and uses an additional Lambda function that processes shadow states and uploads a summary to an Amazon DynamoDB table.

Note
If necessary, you can troubleshoot issues by viewing the AWS Greengrass core logs, particularly router.log:

```
cd /greengrass/ggc/var/log
sudo cat system/router.log | more
```

For more information, see Troubleshooting AWS Greengrass Applications (p. 196).
Configure IAM Roles

Before you begin, make sure that you have completed Module 1 (p. 30) through Module 5 (p. 91). You do not need other components or devices. This module should take about 30 minutes to complete.

**Note**
This module creates and updates a table in DynamoDB – most of the operations are small and fall within the AWS Free Tier. See DynamoDB pricing documentation for more information.

**Configure IAM Roles**

1. Because you are creating a Lambda function that accesses other AWS services, you need to create an IAM role that has access to DynamoDB and AWS Greengrass. For more information about IAM, see the AWS Identity and Access Management documentation.

   In the IAM console, choose Roles, and then choose Create Role:
Choose AWS service, and then choose Greengrass:
Choose the service that will use this role

<table>
<thead>
<tr>
<th>API Gateway</th>
<th>Data Pipeline</th>
<th>IoT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto Scaling</td>
<td>Directory Service</td>
<td>Lambda</td>
</tr>
<tr>
<td>Batch</td>
<td>DynamoDB</td>
<td>Lex</td>
</tr>
<tr>
<td>CloudFormation</td>
<td>EC2</td>
<td>Machine Learning</td>
</tr>
<tr>
<td>CloudHSM</td>
<td>EC2 Container Service</td>
<td>OpsWorks</td>
</tr>
<tr>
<td>CloudWatch Events</td>
<td>EMR</td>
<td>RDS</td>
</tr>
<tr>
<td>CodeBuild</td>
<td>Elastic Beanstalk</td>
<td>Redshift</td>
</tr>
<tr>
<td>CodeDeploy</td>
<td>Elastic Transcoder</td>
<td>S3</td>
</tr>
<tr>
<td>Config</td>
<td>Glue</td>
<td>SMS</td>
</tr>
<tr>
<td>DMS</td>
<td>Greengrass</td>
<td>SNS</td>
</tr>
</tbody>
</table>

Choose **Next: Permissions**.

On the **Attach permissions policies** page, select the following policies: `AWSGreengrassResourceAccessRolePolicy`, `AWSGreengrassFullAccess`, and `AmazonDynamoDBFullAccess`.

Next, choose **Next: Review**. For **Role name**, type `Greengrass_DynamoDB_Role`, and then choose **Create role**.
2. Repeat the prior step to create the role for the AWS Lambda service (instead of the AWS Greengrass service). Give the role the same policies (AWSGreengrassResourceAccessRolePolicy, AWSGreengrassFullAccess, and AmazonDynamoDBFullAccess). For Role name, type Lambda_DynamoDB_Role.

3. In the AWS IoT console, under Greengrass, choose Groups, and choose your AWS Greengrass group. Choose Settings, and then choose Add Role:

The IAM role you just created should appear in the list. If it does not appear, search for it, select it, 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 simulates the passing of a randomized number of cars (from 1 to 20). On every third G light change, the Lambda 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 GitHub repository download the carAggregator.py Lambda function to the car_aggregator folder.
3. Install the boto3 package (AWS SDK for Python) and its dependencies in the car_aggregator folder by running the following command in a command-line window (for Windows, use an elevated command prompt):

   ```bash
   pip install boto3 -t path-to-car_aggregator-folder
   ```

   This results in a directory listing similar to the following:
Greengrass Lambda functions use the AWS SDK to access other Amazon Web Services. For more information, see Boto 3 - The AWS SDK for Python.

4. Compress the contents of the car_aggregator folder into a .zip file named car_aggregator.zip. 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:

   - **Runtime** - choose Python 2.7.
   - **Role** - choose Choose an existing role.
   - **Existing role** - choose Lambda_DynamoDB_Role.

Then, choose Create function.
6. Upload your Lambda function deployment package, as follows:
   a. On the Configuration tab, under Function code, set the following fields:
      • Code entry type - choose Upload a .ZIP file.
      • Runtime - choose Python 2.7.
      • Handler - type 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_Car_Aggregator **. For step-by-step instructions, see the Publish the Lambda function (p. 56) and Create an alias (p. 56) steps in Module 3 (Part 1).

8. In the AWS IoT console, add the Lambda function that you just created to your AWS Greengrass group, as follows:
   
a. On the group configuration page, choose **Lambdas**, and then choose **Add Lambda**:
   
b. Choose **Use existing Lambda**:
   
   ![Add a Lambda to your Greengrass Group](image)
   
   **Add a Lambda to your Greengrass Group**
   
   Local Lambdas are hosted on your Greengrass Core and connected to each other and devices by Subscriptions, but they can also be deployed individually to your Group.
   
   **Create a new Lambda function**
   
   You will be taken to the AWS Lambda Console and can author a new Lambda function.
   
   ![Create new Lambda](image)
   
   **Use an existing Lambda function**
   
   You will choose from a list of existing Lambda functions.
   
   ![Use existing Lambda](image)
   
   c. Choose **GG_Car_Aggregator**, and then choose **Next**:
d. Choose **Alias: GG_CarAggregator**, and then choose **Finish**:

9. **Edit the Lambda function configuration, as follows:**

   a. Choose the ellipsis (…) associated with the Lambda function, then choose **Edit Configuration**:

   b. Under **Lambda lifecycle**, select **Make this function long-lived and keep it running indefinitely**, and then choose **Update**:

**Note**
You can remove other Lambda functions from earlier modules.
Configure Subscriptions

In this step, you create a subscription that enables the GG_TrafficLight shadow to send updated states to the GG_Car_Aggregator Lambda function. This subscription is in addition to the subscriptions that you created in Module 5 (p. 91), 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:
   - Select a source - choose Services and then choose Local Shadow Service.
   - Select a target - choose Lambdas and then choose GG_Car_Aggregator.

Choose Next.

Choose Next, and then choose Finish.

Note
On the Subscriptions page, the target displays the names of the function version and the alias: `GG_Car_Aggregator:GG_CarAggregator`.

The following table shows the complete list of subscriptions that this module requires. The new shadow subscription appears in the last row of the table. You created the other subscriptions in Module 5 (p. 91).
### Configure Subscriptions

<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 or rejected.</td>
</tr>
<tr>
<td>Local Shadow Service</td>
<td>GG_Switch</td>
<td>$aws/things/GG_TrafficLight/shadow/update/rejected</td>
<td>The GG_Switch needs to know whether the update request was accepted or rejected.</td>
</tr>
<tr>
<td>Local Shadow Service</td>
<td>GG_TrafficLight</td>
<td>$aws/things/GG_TrafficLight/shadow/update/delta</td>
<td>The 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>$aws/things/GG_TrafficLight/shadow/update/accepted</td>
<td>The GG_TrafficLight needs to know whether the update request from GG_Switch was accepted or rejected.</td>
</tr>
<tr>
<td>Local Shadow Service</td>
<td>GG_TrafficLight</td>
<td>$aws/things/GG_TrafficLight/shadow/update/rejected</td>
<td>The GG_TrafficLight needs to know whether the update request from GG_Switch was accepted or rejected.</td>
</tr>
</tbody>
</table>

**Note**

Except for Module 5 (p. 91), you can delete the subscriptions from earlier modules that are not included in the table.

4. Make sure that the AWS Greengrass daemon is running, as described in Deploy Cloud Configurations to a Core Device (p. 62).

5. On the group configuration page, from the Actions menu, choose **Deploy** to deploy the updated group configuration to your AWS Greengrass core device.
Test Communications

1. On your computer, open two command-line windows. Just as in Module 5 (p. 91), one window will be for the GG_Switch device and the other for the GG_TrafficLight device.

   **Note**
   These are 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. On every third green light (every 3 minutes), the function handler of the Lambda function is triggered, and a new DynamoDB record is created. After lightController.py and trafficLight.py have run for three minutes, go to the AWS Management Console, search for and open the DynamoDB console. Make sure that the N. Virginia (us-east-1) region is selected, then choose Tables and choose the CarStats table. Next, choose the Items tab:

   ```
   DynamoDB Dashboard
   Tables
   Reserved capacity
   DAX
   Clusters
   Subnet groups
   Parameter groups
   Events
   ```

   You should see entries with basic statistics on cars passed (one entry for every three minutes). You may need to choose the refresh button (two circular arrows) to view updates to the CarStats table:
Note
If necessary, you can troubleshoot issues by viewing the AWS Greengrass core logs, particularly `router.log`:

```
    cd /greengrass/ggc/var/log
    sudo cat system/router.log | more
```

For more information, see Troubleshooting AWS Greengrass Applications (p. 196).

You have reached the end of this tutorial and should now understand the AWS Greengrass programming model and its fundamental concepts, including AWS Greengrass cores, groups, subscriptions, and the deployment process for Lambda functions running at the edge.

You can delete the DynamoDB table, delete the Lambda functions, and stop communications between the AWS Greengrass core device and the AWS IoT cloud. To stop communications, open a terminal on the AWS Greengrass core device and run one of the following commands:

- To shut down the AWS Greengrass core device:

  ```
  sudo halt
  ```

- To stop the AWS Greengrass daemon:
cd /greengrass/ggc/core/
sudo ./greengrassd stop
OTA Updates of AWS Greengrass Core Software

This feature is available for AWS Greengrass Core v1.3.0 and later.

The AWS Greengrass core software comes packaged with an OTA Update Agent that is capable of updating the core's software or the OTA Update Agent itself to the latest respective versions. You can start an update by invoking the CreateSoftwareUpdateJob API or from the Greengrass console. Updating the Greengrass core software provides the following benefits:

- Fix security vulnerabilities.
- Address software stability issues.
- Deploy new or improved features.

An OTA update makes all these benefits available without having to perform the update manually or having the device which is running the core software physically present. The OTA Update Agent also performs a rollback in case of a failed OTA update. Performing an OTA update is optional but can help you manage your AWS Greengrass core devices. Look for announcements of new versions of the core's software on the Greengrass developer forum.

In order to support an OTA update of Greengrass core software by using the OTA Update Agent, your Greengrass core device must:

- Have available local storage three times the amount of the core’s runtime usage requirement.
- Not have trusted boot enabled in the partition containing the Greengrass core platform software. (The AWS 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 containing the Greengrass core platform software.
- Have a connection to the AWS cloud.
- Have a correctly configured AWS Greengrass core and appropriate certificates.

Before launching an OTA Update of Greengrass core software, it is important to note the impact that it will have on the devices in your Greengrass group, both on the core device and on client devices connected locally to that core:

- The core will be shut down during the update.
- Any Lambda functions running on the core will be 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 cloud will be lost and messages routed through the core by client devices will be lost.
- Credential caches will be lost.
- Queues which hold pending work for Lambda functions will be lost.
- Long-lived Lambda functions will lose their dynamic state information and all pending work will be dropped.

The following state information will be preserved during an OTA Update:

- Local shadows
Greengrass OTA Agent

The Greengrass OTA Agent is the software component on the device which handles update jobs created and deployed in the cloud. The Greengrass OTA Agent is distributed in the same software package as the Greengrass core software. The agent is located in ./greengrass/ota/ota_agent/ggc-ota and creates its logs in /var/log/greengrass/ota/ggc_ota.txt.

You can start the Greengrass OTA 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. Once started, the Greengrass OTA Agent will begin listening for Greengrass update jobs from the cloud and execute them sequentially. The Greengrass OTA Agent will ignore all other IoT job types.

Do not start multiple OTA Agent instances as this may cause conflicts.

If your Greengrass core or Greengrass OTA Agent is managed by an init system, see Integration With Init Systems (p. 117) 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 Agent as well as the Greengrass core software. It makes use of the AWS IoT Jobs feature which provides additional commands to manage a Greengrass core software update job. See Jobs for more information on how to manage a Greengrass Update.

The following example shows how to create a Greengrass core software update job using the CLI:

```
aws greengrass create-software-update-job \
  --update-targets-architecture x86_64 \
  --update-targets arn:aws:iot:us-east-1:123456789012:thing/myDevice \
  --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 object containing the job id and job ARN:

```json
{
  "IotJobId": "Greengrass-OTA-c3bd7f36-ee80-4d42-8321-1da0e5b1303",
  "IotJobArn": "arn:aws:iot:us-east-1:123456789012:job/Greengrass-OTA-c3bd7f36-ee80-4d42-8321-1da0e5b1303"
}
```

The create-software-update-job command has the following parameters:

--update-targets-architecture

The architecture of the core device. Must be one of armv7l, 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 which are cores, and the ARNs of thing groups whose members are cores. See IoT thing groups for more information on how to place cores in an IoT thing group.
--update-targets-operating-system

The operating system of the core device. Must be one of ubuntu, amazon_linux or raspbian.

--software-to-update

Specifies whether the core’s software or the OTA Agent software should be updated. Must be one of core or ota_agent.

--s3-url-signer-role

The IAM role which is used to presign the S3 url which links to the Greengrass software update. You must provide a role that has the appropriate policy attached. Here is an example policy document with the minimum required permissions:

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowsIotToAccessGreengrassOTAUpdateArtifacts",
            "Effect": "Allow",
            "Action": ["s3:GetObject"],
            "Resource": [
                "arn:aws:s3:::eu-central-1-greengrass-updates/*",
                "arn:aws:s3:::us-east-1-greengrass-updates/*",
                "arn:aws:s3:::ap-northeast-1-greengrass-updates/*",
                "arn:aws:s3:::us-west-2-greengrass-updates/*",
                "arn:aws:s3:::ap-southeast-2-greengrass-updates/*"
            ]
        }
    ]
}
```

Here is an example Assume Role policy document with the minimum required trusted entities:

```json
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": "sts:AssumeRole",
            "Principal": {
                "Service": "iot.amazonaws.com"
            },
            "Effect": "Allow",
            "Sid": "AllowIotToAssumeRole"
        }
    ]
}
```

--amzn-client-token

[Optional] A client token used to make idempotent requests. Provide a unique token to prevent duplicate updates from being created due to internal retries.

--update-agent-log-level

[Optional] The logging level for log statements generated by the OTA Agent. Must be one of NONE, TRACE, DEBUG, VERBOSE, INFO, WARN, ERROR, or FATAL. The default is ERROR.

Here is an example IAM policy with the minimum permissions required to call the API:
Integration with Init systems

During an OTA update, binaries, some of which may be running, will be updated and restarted. This may cause conflicts if an init system is monitoring the state of either the AWS Greengrass core software or the Greengrass OTA Agent during the update. To help integrate the OTA update mechanism with your monitoring strategies, Greengrass provides the opportunity for user-defined shell scripts to run before and after an update. To tell the OTA agent to run these shell scripts, you must include the `managedRespawn = true` flag in the `./greengrass/config/config.json` file. For example:

```json
{
    "coreThing": {
        ...
    },
    "runtime": {
        ...
    },
    "managedRespawn": true
}
```
When the `managedRespawn` flag is set, the scripts must exist in the directory or the OTA Agent will fail the update. The directory tree should look as follows:

```
<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 Agent prepares to do a self-update, if the `managedRespawn` flag is set to `true` then the OTA Agent will look in the `./greengrass/usr/scripts` directory for the `ota_pre_update.sh` script and run it.

After the OTA Agent completes the update, it will attempt to run the `ota_post_update.sh` script from the `./greengrass/usr/scripts` directory.

**AWS Greengrass Core Update with Managed Respawn**

As the OTA Agent prepares to do an AWS Greengrass core update, if the `managedRespawn` flag is set to `true`, then the OTA Agent will look in the `./greengrass/usr/scripts` directory for the `ggc_pre_update.sh` script and run it.

After the OTA Agent completes the update, it will attempt to run the `ggc_post_update.sh` script from the `./greengrass/usr/scripts` directory.

**Note:**
- 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 will not be run even if present on the device.
- It is imperative that a device which is the target of an update not run two OTA agents for the same AWS IoT thing. Doing so will cause the two OTA Agents to process the same jobs which will lead to conflicts.

**OTA Agent Self-Update**

To perform an OTA Agent self-update follow these steps:
1. Ensure that the AWS Greengrass core is correctly provisioned with valid `config.json` file entries and the necessary certificates.
2. If the OTA Agent is being managed by an init system, ensure that `managedRespawn = true` in the `config.json` file and the scripts `ota_pre_update.sh` and `ota_post_update.sh` are present in the `./greengrass/usr/scripts` directory.
3. Start the ggc-ota agent by running `./greengrass/ota/ota_agent/ggc-ota`. 
4. Create an OTA self update job in the cloud with the CreateSoftwareUpdateJob API (aws greengrass create-software-update-job), making sure the --software-to-update parameter is set to ota_agent.
5. The OTA Agent will perform a self update.

Greengrass Core Software Update

To perform an AWS Greengrass core software update follow these steps:

1. Ensure that the AWS Greengrass core is correctly provisioned with valid config.json file entries and the necessary certificates.
2. If the AWS Greengrass core software is being managed by an init system, ensure that managedRespawn = true in the config.json file and the scripts ggc_pre_update.sh and ggc_post_update.sh are present in the ./greengrass/usr/scripts directory.
3. Start the ggc-ota agent by running ./greengrass/ota/ota_agent/ggc-ota.
4. Create an OTA self update job in the cloud with the CreateSoftwareUpdateJob API (aws greengrass create-software-update-job), making sure the --software-to-update parameter is set to core.
5. The OTA Agent will perform an update of AWS Greengrass core software.
Reset Deployments

This feature is available for AWS Greengrass Core v1.1.0 and later.

You may want to reset a group's deployments in order 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 prior to 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.

**Note**
The Reset Deployments feature is not available in AWS Greengrass Core Software v1.0.0. Also, note that it's not possible to delete a group that has been deployed using v1.0.0.

The `ResetDeployments` command will clean up all deployment information which is stored in the cloud for a given group. It will then instruct 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 `Pending` or `Building`.

```sh
aws greengrass reset-deployments --group-id <GroupId> [--force]
```

**Arguments for the reset-deployments CLI command:**

- `--group-id`
  The group ID.
- `--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 once 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 will perform its clean up operations as well.

The output of the `reset-deployments` CLI command will look like this:

```json
{
  "DeploymentId": "4db95ef8-9309-4774-95a4-eea580b6ceef",
  "DeploymentArn": "arn:aws:greengrass:us-west-2:106511594199:/greengrass/groups/b744ed45-a7df-4227-860a-8d4492ca4342/deployments/4db95ef8-9309-4774-95a4-eea580b6ceef"
}
```

You can check the status of the reset deployment with the `get-deployment-status CLI command`:

```sh
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 will look 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 Greengrass core has not picked up the reset deployment, the DeploymentStatus is InProgress.
Run Lambda Functions on the AWS Greengrass Core

AWS Greengrass provides a containerized Lambda runtime environment for user-defined code. Lambda functions that are deployed to an AWS 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.

SDKs for Greengrass Lambda Functions

AWS provides two SDKs that are used by Greengrass Lambda functions.

**AWS Greengrass Core SDK**

Enables local Lambda functions to interact with local services on the AWS Greengrass core. This SDK is required by all Greengrass Lambda functions. SDK versions are available for common programming languages and platforms.

The following table lists each supported language or platform and the versions of AWS Greengrass core software that it 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>1.0.0 or later</td>
</tr>
<tr>
<td>Java 8</td>
<td>1.1.0 or later</td>
</tr>
<tr>
<td>Node.js 6.10</td>
<td>1.1.0 or later</td>
</tr>
<tr>
<td>C, C++</td>
<td>1.6.0</td>
</tr>
</tbody>
</table>

- Download the AWS Greengrass Core SDK for Python, Java, or Node.js from the Software page of the AWS IoT console.
- Download the AWS Greengrass Core SDK for C from GitHub. This SDK is used by Lambda executables (p. 126).

**AWS SDK**

Enables local Lambda functions to make direct calls to AWS services, such as Amazon S3, DynamoDB, AWS IoT, and AWS Greengrass. To use the AWS SDK in a Greengrass Lambda function, you must include it in your deployment package. When you use the AWS Greengrass Core SDK and the AWS SDK in the same package, 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 SDK from the Getting Started Resource Center.
Migrating Cloud-Based Lambda Functions

The AWS 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 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 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 Greengrass Core SDK supports sending MQTT messages with QoS = 0 only.

This makes it possible for you to test your Lambda functions in the cloud and then migrate them to AWS Greengrass with minimal effort. Lambda executables don't run in the cloud, so you can't use the AWS SDK to test 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 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 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.
Group-Specific Configuration for Greengrass Lambda Functions

AWS Greengrass provides cloud-based management of Greengrass Lambda functions. While a function's code and dependencies are managed using AWS Lambda, AWS Greengrass supports the following group-specific configuration settings:

**Memory limit**

The memory allocation for the function. The default is 16 MB.

**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 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. 125).

**Read access to /sys directory**

Whether the function can access the host's /sys folder. Use this when the function needs to read device information from /sys. The default is false.

**Input payload data type**

The expected encoding type of the input payload for the function, either JSON or binary. Lambda executables (p. 126) support only the binary encoding type (not JSON).

*Note*

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. Support for the binary encoding type is available starting in AWS Greengrass Core Software v1.5.0 and AWS Greengrass Core SDK v1.1.0.

**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. 129) and machine learning resources (p. 154) 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.

Communication Flows for Greengrass Lambda Functions

Greengrass Lambda functions can interact with other members of the AWS Greengrass group, local services, and cloud services (including AWS services).
MQTT Messages

Greengrass Lambda functions can exchange MQTT messages with the following entities:

- Devices in the group.
- Lambda functions in the group.
- AWS IoT.
- Local Device Shadow service.

This communication flow uses a publish-subscribe pattern that's controlled by subscriptions. A subscription defines a message source, message target, and topic filter. Messages that are published to a Lambda function target 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. 188).

Note
Greengrass Lambda functions can exchange messages with devices, other functions, and local shadows when the core is offline, but messages to AWS IoT are queued. For more information, see the section called “MQTT Message Queue” (p. 25).

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 Greengrass core version availability, see Access Local Resources with Lambda Functions (p. 129) and Perform Machine Learning Inference (p. 154).
- Greengrass Lambda functions can use the AWS SDK to communicate with AWS services. For more information, see AWS SDK (p. 122).
- 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.

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 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 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 for 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.

For more information about container reuse, see [Understanding Container Reuse in AWS Lambda](#) on the AWS Compute Blog.

## Lambda Executables

This feature is available for AWS Greengrass Core v1.6.0 only.

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 only the binary encoding type (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 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 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 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 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 that 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 also use the 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 Greengrass console or by using the AWS Greengrass API.
Access Local Resources with Lambda Functions

This feature is available for AWS Greengrass Core v1.3.0 and later.

Developers who use AWS Greengrass can author AWS Lambda functions in the cloud and deploy them to core devices for local execution. On Greengrass cores running Linux, these locally deployed Lambda functions 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. 131)
- How to Configure Local Resource Access Using the AWS Management Console (p. 135)

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 v1.6.0 only. For additional requirements, see the section called “Volume Resources Under the /proc Directory” (p. 130).

**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.

**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/mem).
- GPUs used to accelerate machine learning using on-board GPUs (for example, /dev/nvidia0).
- Cameras used to capture images and videos (for example, /dev/video0).

**Note**

/dev/shm is an exception. It can be configured as a volume resource only. Resources under /dev/shm must be granted r+w permission.
AWS Greengrass also supports resource types that are used to perform machine learning inference. For more information, see Perform Machine Learning Inference (p. 154).

Requirements

The following requirements apply to configuring secure access to local resources:

- You must be using AWS Greengrass Core Software v1.3.0 or later. To create resources for the host's /proc directory, you must be using v1.6.0.
- 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 Greengrass Core Software v1.6.0.
- You can allow read-only access for Lambda functions, but not read-write access. This level of access is managed by AWS 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. 130).
- 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 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 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 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 Greengrass Core v1.3.0 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 (p. 129).*

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 Greengrass (p. 29).*

For a tutorial that uses the AWS Management Console, see *How to Configure Local Resource Access Using the AWS Management Console (p. 135).*

---

### Create Local Resources

First, you use the `create-resource-definition` 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": ""
                        }
                    }
                }
            },
            {
                "Id": "data-device",
                "Name": "TestCamera",
                "ResourceDataContainer": {
                    "LocalDeviceResourceData": {
                        "SourcePath": "/dev/video0",
                        "GroupOwnerSetting": {
                            "AutoAddGroupOwner": true,
                            "GroupOwner": ""
                        }
                    }
                }
            }
        ]
    }
}'
```

---

131
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:

```bash/aws greengrass create-function-definition --cli-input-json '{
  "Name": "MyFunctionDefinition",
  "Resources": [
    {
      "Name": "MyLocalVolumeResource",
      "LastUpdatedTimestamp": "2017-11-15T01:18:42.153Z",
      "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"
    }
  ]
}'
```
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-9fa4-7e8d98c3cf5" \
--resource-definition-version-arn "arn:aws:greengrass:us-west-2:123456789012:/greengrass/definition/resources/db6bf40b-29d3-4c4e-9574-21ab7d74316c/versions/31d0010f-e19a-4c4c-8098-68b79906f87" \
```

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.
Add the Lambda Function to the Group

--function-definition-version-arn "arn:aws:greengrass:us-west-2:123456789012:/greengrass/definition/functions/d1123830-da38-4c4c-a4b7-e92ec7b6d3e/versions/a2e90400-caae-4fdd-b23a-db1892a33c78" \
--subscription-definition-version-arn "arn:aws:greengrass:us-west-2:123456789012:/greengrass/definition/subscriptions/7a8ef3d8-1de3-426c-9554-5b55a32fbc6b/versions/470c858c-7eb3-4add-9d48-230236bfbf6a"

A new group version is returned:

```json
{
    "Version": "291917fb-ec54-4895-823e-27b52da25481",
    "CreationTimestamp": "2017-11-22T01:47:22.487Z",
    "Id": "b36a3aeab-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
# lraTest.py
# 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/LRA/test". 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/LRA/test'

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.
')
        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
```

These commands are provided by the Greengrass API to create and manage resource definitions and resource definition versions:

- CreateResourceDefinition
- CreateResourceDefinitionVersion
- DeleteResourceDefinition
Troubleshooting

Q: Why does my Greengrass group deployment fail with an error similar to:

```bash
group config is invalid:
    ggc_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. 130) 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:

```bash
[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 \"/
greengrass/ggc/packages/1.3.0/rootfs_sys\" at \"/greengrass/ggc/packages/1.3.0/
rootfs_sys/run\"
caused \"invalid argument\"
"
```

A: AWS 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:

```bash
[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 \"/
greengrass/ggc/packages/1.3.0/rootfs_sys\" at \"/greengrass/ggc/packages/1.3.0/
rootfs_sys/dev/shm\"
caused \"operation not permitted\"
"
```

A: `/dev/shm` can only be configured as read/write. Change the resource permission to `rw` to resolve the issue.

How to Configure Local Resource Access Using the AWS Management Console

This feature is available for AWS Greengrass Core v1.3.0 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 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 (p. 129).

This tutorial describes how to use the AWS Management Console to configure access to local resources that are present on an AWS Greengrass core device. It contains the following high-level steps:

1. Create a Lambda Function Deployment Package (p. 136)
2. Create and Publish a Lambda Function (p. 138)
3. Add the Lambda Function to the Group (p. 142)
4. Add a Local Resource to the Group (p. 145)
5. Add Subscriptions to the Group (p. 147)
6. Deploy the Group (p. 149)

For a tutorial that uses the AWS Command Line Interface (CLI), see How to Configure Local Resource Access Using the AWS Command Line Interface (p. 131).

Prerequisites

To complete this tutorial, you need:

- A Greengrass group and a Greengrass core (v1.3.0 or later). To learn how to create a Greengrass group or core, see Getting Started with AWS Greengrass (p. 29).
- The following directories created on the Greengrass core device:
  - /src/LRAtest
  - /dest/LRAtest

The owner group of these directories must have read and write access to the directories. For example, you might use the following command to grant access:

```bash
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 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.

```python
# lraTest.py
# 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 Lambda Function Deployment Package

# 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 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 Greengrass Core SDK Python 2.7 version 1.2.0, as follows:

   a. In the AWS IoT console, in the left pane, choose **Software**.

   ![AWS IoT Console](image)

   Monitor
   Onboard
   Manage
   Greengrass
   Secure
   Defend
   Act
   Test

   ![Software](image)

   Settings
   Learn

   b. Under **SDKs**, for **AWS Greengrass Core SDK**, choose **Configure download**.
c. Choose **Python 2.7 version 1.2.0**, and then choose **Download Greengrass Core SDK**.

3. Unpack the `greengrass-core-python-sdk-1.2.0.tar.gz` file.

   **Note**
   For ways that you can do this on different platforms, see this step (p. 51) in the Getting Started section. For example, you might use the following `tar` command:

   ```bash
tar -xzf greengrass-core-python-sdk-1.2.0.tar.gz
   ```

4. Open the extracted `aws_greengrass_core_sdk/sdk` folder, and unzip `python_sdk_1_2_0.zip`.

5. Zip the following items into a file named `lraTestLambda.zip`:
   - `lraTest.py`. App logic.
   - `greengrasssdk`. Required library for all Python Lambda functions.
   - `Greengrass AWS SW License (IoT additiona) vr6.txt`. Required Greengrass Core Software License Agreement.

   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 **Author from scratch** section, use the following values:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>TestLRA</td>
</tr>
<tr>
<td>Runtime</td>
<td>Python 2.7</td>
</tr>
<tr>
<td>Role</td>
<td>Create new role from template(s)</td>
</tr>
<tr>
<td>Role name</td>
<td>Greengrass_role_does_not_matter</td>
</tr>
</tbody>
</table>

   *(This role isn't used by AWS Greengrass.)*

5. At the bottom of the page, choose **Create function**.
Now, upload your Lambda function deployment package and register the handler.

1. On the Configuration tab for the TestLRA function, in Function code, use the following values:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code entry type</td>
<td>Upload a .ZIP file</td>
</tr>
<tr>
<td>Runtime</td>
<td>Python 2.7</td>
</tr>
<tr>
<td>Handler</td>
<td>lraTest.function_handler</td>
</tr>
</tbody>
</table>

2. Choose Upload.
3. Choose your lraTestLambda.zip deployment package.
4. At the top of the page, choose Save.

Tip
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.

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, type First version, and then choose Publish.
3. On the TestLRA: 1 configuration page, from the Actions menu, choose Create alias.
4. On the Create a new alias page, use the following values:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>test</td>
</tr>
</tbody>
</table>
### Add the Lambda Function to the Group

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>1</td>
</tr>
</tbody>
</table>

**Note**

AWS Greengrass doesn't support Lambda aliases for `$LATEST` versions.

5. **Choose Create.**

An alias is a pointer to one or two versions. Select the version(s) you would like the alias to point to.

- **Name**
  - test

- **Description**

- **Version**
  - 1

You can shift traffic between two versions, based on weights (%) that you assign. Click here to learn more.

- **Additional Version**

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 TestLRA 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**.
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.

1. On the Lambdas page, choose the TestLRA Lambda function.

2. On the TestLRA configuration page, choose Edit.
3. On the Group-specific Lambda configuration page, use the following values:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timeout</td>
<td>30 seconds</td>
</tr>
<tr>
<td>Lambda lifecycle</td>
<td>Make this function long-lived and keep it running indefinitely</td>
</tr>
</tbody>
</table>

For more information, see the section called “Lifecycle Configuration” (p. 125).
4. 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 a Greengrass group and grant the function read and write access to the resource. A local resource has a group-level scope, which makes it accessible by all Lambda functions in the group.

1. On the group configuration page, choose Resources.

2. On the Local Resources tab, choose Add local resource.
3. On the Create a local resource page, use the following values:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource name</td>
<td>testDirectory</td>
</tr>
<tr>
<td>Resource type</td>
<td>Volume</td>
</tr>
</tbody>
</table>
Add a Local Resource to the Group

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source path</td>
<td>/src/LRAtest</td>
</tr>
<tr>
<td>(This path must exist on the host OS.)</td>
<td></td>
</tr>
<tr>
<td>Destination path</td>
<td>/dest/LRAtest</td>
</tr>
<tr>
<td>(This path must exist on the host OS.)</td>
<td></td>
</tr>
<tr>
<td>Group owner file access permission</td>
<td>Automatically add OS group permissions of the Linux group that owns the resource</td>
</tr>
</tbody>
</table>

The **Source path** is the local absolute path of the resource on the file system of the core device. This path can't start with /sys.

The **Destination path** is the absolute path of the resource in the Lambda namespace.

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. 130).

4. Under **Lambda function affiliations**, choose **Select**.
5. Choose **TestLRA**, choose **Read and write access**, and then choose **Done**.
Add Subscriptions to the Group

At the bottom of the page, choose Save. The Resources page displays the new testDirectory resource.

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 Greengrass.

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, type LRA/test, and then choose Next.
Add Subscriptions to the Group


Next, configure a subscription that invokes the function from AWS IoT.

1. On the Subscriptions page, 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 Services, and then choose IoT Cloud.
   b. For Select a target, choose Lambdas, and then choose TestLRA.
   c. Choose Next.
3. On the Filter your data with a topic page, for Optional topic filter, type invoke/LRAFunction, and then choose Next.
4. Choose **Finish**. The **Subscriptions** page displays both subscriptions.

**Step 6: Deploy the AWS Greengrass Group**

In this step, you deploy the current version of the group definition.

1. Make sure that the AWS 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/1.6.0/bin/daemon`, then the daemon is running.

   **Note**
   
   The version in the path depends on the AWS 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 Greengrass also supports manually specified endpoints. You're only prompted for the discovery method the first time that the group is deployed.

**Automatic detection**
Greengrass will detect and override connection information as it changes.

**Note**
If prompted, grant permission to create the AWS Greengrass service role on your behalf, which allows AWS Greengrass to access other AWS services. You need to do this only one time per account.

The **Deployments** page shows the deployment timestamp, version ID, and status. When completed, the deployment should show a **Successfully completed** status.

For help troubleshooting any issues that you encounter, see *Troubleshooting AWS Greengrass Applications* (p. 196).

**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, use the following values:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subscription topic</td>
<td>LRA/test</td>
</tr>
<tr>
<td>MQTT payload display</td>
<td>Display payloads as strings</td>
</tr>
</tbody>
</table>

3. Choose **Subscribe to topic**. Your Lambda function publishes to the LRA/test topic.
4. In the **Publish** section, type **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.
You can see the test file that the Lambda creates by looking in the /src/LRAtest directory on the Greengrass core device. Although the Lambda 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. However, any changes to the destination path are reflected in the source path on the actual file system.

For help troubleshooting any issues that you encounter, see *Troubleshooting AWS Greengrass Applications* (p. 196).
Perform Machine Learning Inference

This feature is available for AWS Greengrass Core v1.5.0 and later.

With AWS Greengrass, you can perform machine learning (ML) inference at the edge on locally generated data using cloud-trained models. This lets 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. 159).

How AWS 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 Greengrass ML inference workflow.

AWS 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. 129).
Machine Learning Resources

Machine learning resources represent cloud-trained inference models that are deployed to an AWS 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 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 Greengrass checks the source for changes. If changes are detected, then AWS Greengrass updates the local model.

Supported Model Sources

AWS 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 Greengrass ML inference currently supports only SSE-S3 or SSE-KMS encryption options. 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 region support must be available, as follows:
  - Amazon SageMaker model sources are supported only in the following regions: US East (N. Virginia), US West (Oregon), and Asia Pacific (Tokyo).
  - Amazon S3 model sources are supported only in regions that have both AWS Greengrass support and Amazon S3 support.
- AWS Greengrass must have read permission to the model source, as described in the following sections.

Amazon SageMaker

AWS 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 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 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"
```
Note

Amazon SageMaker is a fully managed ML service that enables you 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 Greengrass supports models that are stored in Amazon S3 as tar.gz or .zip files.

To enable AWS Greengrass to access models that are stored in Amazon S3 buckets, you must grant AWS 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"
    }
  ]
}
```

For more information, see Embedding Inline Policies in the IAM User Guide.

Requirements

The following requirements apply for creating and using machine learning resources:

- You must be using AWS Greengrass Core Software v1.5.0 or later.
- Access to the local destination directory where the resource is stored must not require root privileges.
- Lambda functions can't perform privileged operations on the resource. Only read or read and write permissions are available.
- 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:_-]+.
Precompiled Libraries for ML Frameworks

To help you quickly get started experimenting with ML inference, AWS Greengrass provides precompiled libraries for the following ML frameworks.

- **Apache MXNet** (Apache License 2.0)
- **TensorFlow** (Apache License 2.0)
- **Chainer** (MIT License)

The precompiled MXNet and TensorFlow libraries can be installed on NVIDIA Jetson TX2, Intel Atom, and Raspberry Pi platforms. The libraries are available from the Software page of the AWS IoT console. 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.

**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.

**Note**

We recommend using MXNet v0.11 for AWS Greengrass ML inference. To configure Amazon SageMaker to train models using the recommended version, see How to Configure Amazon SageMaker to Use MXNet v0.11 (p. 158).

**TensorFlow Model-Serving Limitations on Raspberry Pi**

TensorFlow officially only supports installation on 64-bit laptop or desktop operating systems. Therefore, the precompiled TensorFlow libraries that AWS Greengrass provides for 32-bit ARM platforms (such as Raspberry Pi) have inherent limitations and are intended for experimentation purposes only.

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 Amazon SageMaker to Use MXNet v0.11

This section describes how to configure Amazon SageMaker to train models using MXNet v0.11. This is the recommended version for AWS Greengrass ML inference.

1. Create an Amazon S3 bucket by following the Create a Bucket procedure in the Amazon Simple Storage Service Console User Guide.

   Make sure that you include `sagemaker` in the name (for example, `sagemaker-datetime`).

2. Create an Amazon SageMaker notebook instance by following the Create an Amazon SageMaker Notebook Instance procedure in the Amazon SageMaker Developer Guide.

3. When the status of the notebook instance is InService, choose the Open action for your notebook instance.

4. Choose New, and then choose `conda_mxnet_p27` from the list of Jupyter kernels. This opens an MXNet environment for Python 2.7.

5. Override the default `sagemaker.mxnet.MXNet.train_image` field in the code with an MXNet v0.11 container image, as follows:

   a. In the following code, replace the `container-image` placeholder value with the MXNet v0.11 container image that you want to use:

      - For MXNet v0.11 on Python2.7 with CPU, specify `sagemaker-mxnet-py2-cpu`.
      - For MXNet v0.11 on Python2.7 with GPU, specify `sagemaker-mxnet-py2-gpu`.
      - For MXNet v0.11 on Python 3.6 with CPU, specify `sagemaker-mxnet-py3-cpu`.
      - For MXNet v0.11 on Python 3.6 with GPU, specify `sagemaker-mxnet-py3-gpu`.

         ```python
         region = sagemaker_session.boto_session.region_name
         mnist_estimator.train_image = lambda: '{}.dkr.ecr.{}.amazonaws.com/container-image:1.0'.format(region)
         ```

   b. Insert the code before the call to `sagemaker.mxnet.MXNet.fit`, which sends a CreateTrainingJob request to Amazon SageMaker using the MXNet v0.11 container image. This overrides the default v0.12 image with the v0.11 image.

      The following shows a sample MXNet training code snippet with the new code highlighted.

      ```python
      ```
Now you can continue training the model as described in the Amazon SageMaker documentation.

How to Configure Machine Learning Inference Using the AWS Management Console

This feature is available for AWS Greengrass Core v1.5.0 and 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. 154).

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:

1. Configure the Raspberry Pi (p. 160)
2. Install the MXNet Framework (p. 160)
3. Create a Model Package (p. 162)
4. Create and Publish a Lambda Function (p. 162)
5. Add the Lambda Function to the Group (p. 167)
6. Add Resources to the Group (p. 169)
7. Add a Subscription to the Group (p. 172)
8. Deploy the Group (p. 173)
Prerequisites

To complete this tutorial, you need:

- Raspberry Pi 3 Model B.
- 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 Greengrass (p. 29). The Getting Started section also includes steps for installing the AWS Greengrass Core software on a Raspberry Pi.

Note
This tutorial uses a Raspberry Pi, but AWS Greengrass supports other platforms, such as Intel Atom and NVIDIA Jetson TX2 (p. 178).

Step 1: Configure the Raspberry Pi

In this step, you update 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. Update Raspbian Jessie.

```
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.

```
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.
Install the MXNet Framework

**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 “Precompiled Libraries for ML Frameworks” (p. 157).

1. On your computer, open the AWS IoT console.
2. In the left pane, choose Software.

![AWS IoT](image)

Monitor
Onboard
Manage
Greengrass
Secure
Defend
Act
Test

**Software**
Settings
Learn

3. In the Machine learning libraries section, for MXNet/TensorFlow precompiled libraries, choose Configure download.
4. On the Machine learning libraries page, under Software configurations, for MXNet Raspberry Pi version 0.11.0, choose Download.

**Note**
By downloading this software you agree to the Apache License 2.0.

5. Transfer the downloaded `ggc-mxnet-v0.11.0-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. 48) in the Getting Started section. For example, you might use the following `scp` command:
6. In your Raspberry Pi terminal, unpack the transferred file.

```
tar -xzf ggc-mxnet-v0.11.0-python-raspi.tar.gz
```

7. Install the MXNet framework.

```
./mxnet_installer.sh
```

**Note**
You can continue to the section called “Create a Model Package” (p. 162) while the framework is installing, but you must wait for the installation to complete before proceeding to the section called “Create and Publish a Lambda Function” (p. 162).

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 \(\text{ok}\). If all tests are successful, the final log statement contains \(\text{OK}\). Note that running the unit tests increases the installation time.

The script also creates a Lambda function deployment package named \(\text{greengrassObjectClassification.zip}\). This package contains the function code and dependencies, including the \text{mxnet} Python module that Greengrass Lambda functions need to work with MXNet models. You upload this deployment package later.

8. When the installation is complete, transfer \(\text{greengrassObjectClassification.zip}\) to your computer. Depending on your environment, you can use the \text{scp} command or a utility such as \text{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 \text{.zip} file. AWS Greengrass can use models from Amazon S3, provided that they use the \text{tar.gz} or \text{.zip} format.

1. Download the following files to your computer:

   - \text{squeezenet_v1.1-0000.params}. A parameter file that describes weights of the connectivity.
   - \text{squeezenet_v1.1-symbol.json}. A symbol file that describes the neural network structure.
   - \text{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 \text{squeezenet.zip}. You upload this model package to Amazon S3 in the section called “Add Resources to the Group” (p. 169).

### Step 4: Create and Publish a Lambda Function

In this step, you create a Lambda function and configure it to use the deployment package that was created in Step 2: Install the MXNet Framework (p. 160). Then, you publish a function version and create an alias.
The Lambda function deployment package is named `greengrassObjectClassification.zip`. 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:**
  - `greengrasssdk`. Required library for all Python Lambda functions.
  - `mxnet`. Required library for Python Lambda functions that run local inference using MXNet.

- **License:**
  - `license`. Contains the required Greengrass Core Software License Agreement.

**Note**
You can reuse these dependencies and license when you create new MXNet inference Lambda functions.

First, create the Lambda function.

1. In the AWS IoT console, in the left 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 takes you to the AWS Lambda console.

5. Choose Author from scratch.

6. In the Author from scratch section, use the following values:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>greengrassObjectClassification</td>
</tr>
<tr>
<td>Runtime</td>
<td>Python 2.7</td>
</tr>
<tr>
<td>Role</td>
<td>Create new role from template(s)</td>
</tr>
<tr>
<td>Role name</td>
<td>Greengrass_role_does_not_matter</td>
</tr>
</tbody>
</table>

(This role isn't used by AWS Greengrass.)

7. At the bottom of the page, choose Create function.
Now, upload your Lambda function deployment package and register the handler.

1. On the Configuration tab for the `greengrassObjectClassification` function, use the following values for Function code:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code entry type</td>
<td>Upload a .ZIP file</td>
</tr>
<tr>
<td>Runtime</td>
<td>Python 2.7</td>
</tr>
<tr>
<td>Handler</td>
<td><code>greengrassObjectClassification.function_handler</code></td>
</tr>
</tbody>
</table>

2. Choose Upload.

3. Choose your `greengrassObjectClassification.zip` deployment package.
4. At the top of the page, 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**.

![Publish new version](image)

2. For **Version description**, type **First version**, and then choose **Publish**.

3. On the **greengrassObjectClassification: 1** configuration page, from the **Actions** menu, choose **Create alias**.

![Create alias](image)

4. On the **Create a new alias** page, use the following values:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>mlTest</td>
</tr>
<tr>
<td>Version</td>
<td>1</td>
</tr>
</tbody>
</table>

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

5. Choose **Create**.
Add the Lambda Function to the Group

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.

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. On the **Use existing Lambda** page, 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 of the Lambda function.

1. On the **Lambdas** page, choose the `greengrassObjectClassification` Lambda function.

2. On the `greengrassObjectClassification` configuration page, choose **Edit**.
3. On the **Group-specific Lambda configuration** page, use the following values:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory limit</td>
<td>96 MB</td>
</tr>
<tr>
<td>Timeout</td>
<td>10 seconds</td>
</tr>
<tr>
<td>Lambda lifecycle</td>
<td>Make this function long-lived and keep it running indefinitely</td>
</tr>
<tr>
<td>Read access to /sys directory</td>
<td>Enable</td>
</tr>
</tbody>
</table>

For more information, see the section called “Lifecycle Configuration” (p. 125).
4. At the bottom of the page, 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 enables 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 (p. 129).

1. On the group configuration page, choose Resources.

2. On the Local Resources tab, choose Add local resource.

3. On the Create a local resource page, use the following values:
### Add Resources to the Group

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource name</td>
<td>videoCoreSharedMemory</td>
</tr>
<tr>
<td>Resource type</td>
<td>Device</td>
</tr>
<tr>
<td>Device path</td>
<td>/dev/vcsm</td>
</tr>
<tr>
<td>Group owner file access permission</td>
<td>Automatically add OS group permissions of the Linux group that owns the resource</td>
</tr>
</tbody>
</table>

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`.

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. 130).

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. At the bottom of the page, choose **Add another resource**.
7. On the **Create a local resource** page, use the following values:
Add Resources to the Group

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource name</td>
<td>videoCoreInterface</td>
</tr>
<tr>
<td>Resource type</td>
<td>Device</td>
</tr>
<tr>
<td>Device path</td>
<td>/dev/vchiq</td>
</tr>
<tr>
<td>Group owner file access permission</td>
<td>Automatically add OS group permissions of the Linux group that owns the resource</td>
</tr>
</tbody>
</table>

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 **Machine Learning** tab, choose **Add machine learning resource**.
2. On the **Create a machine learning resource** page, for **Resource name**, type **squeezenet_model**.
3. For **Model source**, choose **Locate or upload a model in S3**.
4. Under **Model from S3**, choose **Select**, and then choose **Create S3 bucket**.
5. For **Bucket name**, type a name that contains the string `greengrass` (such as `greengrass-datetime`), and then choose **Create**.

   **Note**
   Don't use a period (".") in the bucket name.

6. Choose **Upload a model**, and then choose the `squeezenet.zip` package that you created in the section called “Create a Model Package” (p. 162).

7. For **Local path**, type `/greengrass-machine-learning/mxnet/squeezenet`.

   This is the destination for the local model in the Lambda runtime namespace. When you deploy the group, AWS 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).

8. Under **Lambda function affiliations**, choose **Select**.

9. Choose **greengrassObjectClassification**, choose **Read-only access**, and then choose **Done**.

10. At the bottom of the page, 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 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. 155).

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**, type 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 the **Optional topic filter** field, type **hello/world**, and then choose **Next**.

4. 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 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/1.6.0/bin/daemon`, then the daemon is running.

   **Note**
   The version in the path depends on the AWS 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 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 AWS Greengrass service role on your behalf, which allows AWS Greengrass to access other AWS services. You need to do this only one time per account.
The **Deployments** page shows the deployment time stamp, version ID, and status. When completed, the deployment should show a **Successfully completed** status.

For help troubleshooting any issues that you encounter, see *Troubleshooting AWS Greengrass Applications* (p. 196).

**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. On the AWS IoT console home page, choose **Test**.

2. For **Subscriptions**, use the following values:
3. **Choose **Subtract 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.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subscription topic</td>
<td>hello/world</td>
</tr>
<tr>
<td>MQTT payload display</td>
<td>Display payloads as strings</td>
</tr>
</tbody>
</table>

Troubleshooting **AWS 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.

```
sudo su
```

2. Navigate to the /log directory.

```
cd /greengrass/ggc/var/log
```

3. **Check runtime.log or python_runtime.log.**

For more information, see the section called “Troubleshooting with Logs” (p. 199).

"Unpacking" Error in runtime.log

If runtime.log contains an error similar to the following, ensure 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`,
error: open `/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, try repackaging the model using the following command:

```bash
tar -zcvf model.tar.gz ./model
```

For example:

```bash
# 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 running the command.

   ```bash
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. 162).
   Also 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:

   ```bash
   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 running the command.

   **Note**
   This directory and its contents are in the Lambda runtime namespace, so they aren't visible in a regular Linux namespace.

   ```bash
   sudo nsenter -t pid -m /bin/bash
   ```
3. List the contents of the local directory that you specified for the ML resource.

```bash
cd /greengrass-machine-learning/mxnet/squeezenet/
ls -ls
```

You should see the following files:

- `synset.txt` (31675 bytes)
- `squeezenet_v1.1-symbol.json` (28707 bytes)
- `squeezenet_v1.1-0000.params` (4945062 bytes)

Next Steps

Next, explore other inference apps. AWS 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. 160).

Configuring an NVIDIA Jetson TX2

To run this tutorial on the GPU of an NVIDIA Jetson TX2, you must add additional local device resources and configure access for the Lambda function.

**Note**

Your Jetson must be configured before you can install the AWS Greengrass Core software. For more information, see Configuring NVIDIA Jetson TX2 for AWS Greengrass (p. 40).

1. Add the following local device resources. Follow the procedure in Add Resources to the Group (p. 169).

   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-db-gpu</td>
<td>/dev/nvhost-db-gpu</td>
</tr>
<tr>
<td>nvhost-prof-gpu</td>
<td>/dev/nvhost-prof-gpu</td>
</tr>
<tr>
<td>nvmmap</td>
<td>/dev/nvmap</td>
</tr>
</tbody>
</table>

2. Edit the configuration of the Lambda function to increase **Memory limit** to 1000 MB. Follow the procedure in Add the Lambda Function to the Group (p. 167).
Greengrass Discovery RESTful API

All devices that communicate with an AWS Greengrass core must be a member of a Greengrass group. Each group must have an AWS Greengrass core. The Discovery API enables devices to retrieve information required to connect to an AWS Greengrass core that is in the same Greengrass group as the device. When a device first comes online, it can connect to the AWS Greengrass cloud service and use the Discovery API to find:

- The group to which it belongs.
- The IP address and port for the AWS Greengrass core in the group.
- The group's root CA certificate, which can be used to authenticate the AWS Greengrass core device.

To use this API, send HTTP requests to the following URI:

```
https://greengrass.iot.aws-region.amazonaws.com/greengrass/discover/thing/thing-name
```

Use port 8443 when connecting. For a list of supported regions and endpoints for the AWS 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:

```
HTTP GET https://greengrass.iot.aws-region.amazonaws.com/greengrass/discover/thing/thing-name
```

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. 180).

Authorization

Retrieving the connectivity information requires a policy that allows the caller to perform the 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:
Example Discover Response Documents

The following document shows the response for a device that is a member of a group with one AWS 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 Greengrass core, multiple endpoints, and multiple group CAs:

```
{
  "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-2
-----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 Greengrass group must define exactly one AWS Greengrass core. Any response from the AWS Greengrass cloud service that contains a list of AWS Greengrass cores only contains one AWS Greengrass core.
Use Greengrass OPC-UA to Communicate with Industrial Equipment

Greengrass supports OPC-UA, an information exchange standard for industrial communication. OPC-UA allows you to ingest and process messages from industrial equipment and deliver them to devices in your Greengrass group or to the cloud based on rules you define.

The Greengrass implementation of OPC-UA supports certificate-based authentication. It is based on an open-source implementation, and is fully customizable. You can also bring your own implementation of OPC-UA, and implement your own support for other custom, legacy, and proprietary messaging protocols.

In this section we will cover the following steps:

- Connect to an existing OPC-UA server.
- Monitor an existing OPC-UA node within that server.
- Get called back when the monitored node’s value changes.

Architectural Overview

Greengrass implements OPC-UA as a Lambda function in NodeJS. Since Lambda functions running on Greengrass cores have access to network resources, you can create Lambda functions that proxy information from your existing OPC-UA servers over TCP to other functions or services in your Greengrass group.

You can configure Greengrass to have a long-lived connection to your OPC-UA server(s), and, using OPC-UA Subscriptions, you can have your OPCUA_Adapter Lambda function monitor changes to predefined nodes. Any change to those nodes triggers a Publish event from the OPC-UA server, which will be received by your Lambda function, and republished into predefined topic names.

The topic structure is constructed as follows:
Set Up a Test OPC-UA Server

Use the following commands to set up a test OPC-UA server. Or, if you already have an OPC-UA server you'd like to use instead, you may skip this step.

```bash
git clone git://github.com/node-opcua/node-opcua.git
cd node-opcua
git checkout v0.0.64
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_id>4840/UADiscovery
err Cannot find module 'usage'
skipping installation of cpu_usage and memory_usage nodes
server on port      : 26543
endpointUrl         : opc.tcp://<your_instance_id>us-west-2.compute.internal:26543
serverInfo          : 
  applicationUri                  : urn:54f7890cca4c49a1:NodeOPCUA-Server
  productUri                      : NodeOPCUA-Server
  applicationName                 : locale=en text=NodeOPCUA
  applicationType                 : SERVER
  gatewayServerUri                : null
  discoveryProfileUri             : null
  discoveryUrls                    :
  productName                     : NODEOPCUA-SERVER
  buildInfo           :
    productUri                  : NodeOPCUA-Server
    manufacturerName   : Node-OPCUA : MIT Licence ( see http://node-opcua.github.io/)
    productName                 : NODEOPCUA-SERVER
    softwareVersion             : 0.0.65
    buildNumber                 : 1234
    buildDate                   : Thu Aug 03 2017 00:13:50 GMT+0000 (UTC)

server now waiting for connections. CTRL+C to stop
```

Make sure your Greengrass Group is ready

- Create a Greengrass group (find more details in Configure AWS Greengrass on AWS IoT (p. 43).)
- Set up a Greengrass Core on one of the supported platforms (Raspberry-pi for example (p. 30))
Use Greengrass OPC-UA to Interact with your 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:** This Lambda function uses the node-opcua library (v0.0.64), which attempts to re-generate some model files at runtime. That doesn't work when running as a Lambda function on Greengrass, because Lambda functions start with a Read-Only file system, so any code trying to generate other code would not work. The next step fixes this.

2. Change the file at `node_modules/node-opcua/lib/misc/factories.js`: line 109 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 Lambda function to contain the server IP and Port that you want to connect to, as well as the node IDs you would like to monitor. By default it comes with the following example configuration:

```javascript
const configSet = {
  server: {
    name: 'server',
    url: 'opc.tcp://localhost:26543',
  },
  subscriptions: [ { name: 'MyPumpSpeed', nodeId: 'ns=1;s=PumpSpeed', }, ],
};
```

In this case, we are connecting to an OPC-UA server running on the same host as our Greengrass Core, on port 26543, and monitoring one node that has an 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 allows you to specify username/password or no authentication.
Here is how 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` and `securityModes`, `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. Upload your Lambda

Create a Greengrass Lambda function. You can find more details on how to do that in Configure the Lambda Function for AWS Greengrass (p. 57). In a nutshell, create a Lambda function code archive by doing the following:

```
# Download the nodejs greengrass sdk from

# Install Greengrass SDK in the node_modules directory
tar -zxvf aws-greengrass-core-sdk-js-*.tar.gz -C /tmp/
unzip /tmp/aws_greengrass_core_sdk_js/sdk/aws-greengrass-core-sdk.zip -d node_modules

# Archive the whole directory as a zip file
zip -r opcuaLambda.zip * -x \*.git\*

# Create an AWS Lambda with the created zip
aws lambda create-function --function-name <Function_Name> --runtime 'nodejs6.10' --role <Your_Role> --handler 'index.handler' --zip-file opcuaLambda.zip
```

Add this Lambda to your Greengrass Group. Details are, again, in: Configure the Lambda Function for AWS Greengrass (p. 57).

6. Configure and Deploy the Lambda function to your Greengrass Group

After creating your AWS Lambda function, you add it to your Greengrass Group. Follow the instructions in same section as above.

- Make sure to specify the Lambda function as Long-Running.
- Give it at least 64MB of memory size.

You can now create a deployment with your latest configuration. You can find details in Deploy Cloud Configurations to an AWS Greengrass Core Device (p. 62).
Verify that your Lambda function is receiving OPC-UA Publishes and posting them onto Greengrass

As described in the Architecture section (p. 182), your Lambda function should start receiving messages from your OPC-UA server. If you are using your own custom OPC-UA server, make sure you trigger a change in the OPC-UA node Id you specified, so that you see the change received by your Lambda function. If you are using the example server above, the PumpSpeed node is configured to simulate a series of consecutive updates, so you should expect your Lambda function to receive multiple messages a second.

You can see messages received by your Lambda function in one of two ways:

- Watch the Lambda function's logs

You can view the logs from your Lambda function by running the following command:

```
sudo cat ggc/var/log/user/us-west-2/your_account_id/your_function_name.log
```

The logs should look similar to:

```
[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}}"
```

- Configure Greengrass to forward messages from your Lambda function to the IoT Cloud.

Follow the steps outlined in Verify the Lambda Function Is Running on the Device (p. 63) to receive messages on the AWS IoT console.

**Note:**

- Make sure there is a Subscription from your Lambda function going to the IoT Cloud. Details are in Configure the Lambda Function for AWS Greengrass (p. 57).
- Since messages are forwarded to the cloud, make sure you terminate either the example server you configured above, or stop the Greengrass core, so that you don't end up publishing a lot of messages to IoT cloud and getting charged for them!

**Next Steps**

With Greengrass, you can use this same architecture to create your own implementation of OPC-UA, and also implement your own support for custom, legacy, and proprietary messaging protocols. Since Lambda functions running on Greengrass cores have access to network resources, you can use them to implement support for any protocol that rides on top of TCP-IP. In addition, you can also take advantage of Greengrass Local Resource Access to implement support for protocols that need access to hardware adapters/drivers.
AWS Greengrass Security

AWS Greengrass uses X.509 certificates, managed subscriptions, AWS IoT policies, and IAM policies and roles to ensure your Greengrass applications are secure. AWS Greengrass core devices require an AWS IoT thing, a device certificate, and an AWS IoT policy to communicate with the Greengrass cloud service.

This allows AWS Greengrass core devices to securely connect to the AWS IoT cloud services. It also allows the Greengrass cloud service to deploy configuration information, Lambda functions, and managed subscriptions to AWS Greengrass core devices.

AWS IoT devices require an AWS IoT thing, a device certificate, and an AWS IoT policy to connect to the Greengrass service. This allows AWS IoT devices to use the Greengrass Discovery Service to find and connect to an AWS Greengrass core device. AWS IoT devices use the same device certificate used to connect to AWS IoT device gateway and AWS Greengrass core devices. The following diagram shows the components of the AWS Greengrass security model:

A - Greengrass service role
   A customer-created IAM role that allows AWS Greengrass access to your AWS IoT and Lambda resources.

B - Core device certificate
   An X.509 certificate used to authenticate an AWS Greengrass core.

C - Device certificate
   An X.509 certificate used to authenticate an AWS IoT device.

D - Group role
   A role assumed by AWS Greengrass when calling into the cloud from a Lambda function on an AWS Greengrass core.
**Configuring Greengrass Security**

To configure your Greengrass application's security:

1. Create an AWS IoT thing for your AWS Greengrass core device.
2. Generate a key pair and device certificate for your AWS Greengrass core device.
3. Create and attach an AWS IoT policy to the device certificate. The certificate and policy allow the AWS Greengrass core device access to AWS IoT and Greengrass cloud services.
4. Create a Greengrass service role. This IAM role grants AWS Greengrass access to your Greengrass and AWS IoT resources. You only need to create a service role once per AWS account.
5. (Optional) Create a Greengrass group role. This role grants permission to Lambda functions running on an AWS Greengrass core to call other AWS services (in the cloud). You need to do this for each Greengrass group you create.
6. Create an AWS IoT thing for each device that will connect to your AWS Greengrass core.
7. Create device certificates, key pairs, and AWS IoT policies for each device that will connect to your AWS Greengrass core.

**Note**
You can also use existing AWS IoT things and certificates.

**Device Connection Workflow**

This section describes how devices connect to the AWS Greengrass cloud service and AWS Greengrass core devices.

- An AWS 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 Greengrass core device downloads group membership information from the Greengrass service.
- When a deployment is made to the AWS Greengrass core device, the Device Certificate Manager (DCM) handles certificate management for the AWS 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 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 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**

A subscription table is used to define how messages are exchanged within a Greengrass group (between AWS Greengrass core devices, AWS IoT devices, and Lambda functions). Each entry in the subscription
table specifies a source, a destination, and an MQTT topic over which messages are sent/received. Messages can be exchanged only if an entry exists in the subscription table specifying the source (message sender), the target (message recipient), and the MQTT topic. Subscription table entries specify passing messages in one direction, from the source to the target. If you want two-way message passing, create two subscription table entries, one for each direction.

**MQTT Core Server Certificate Rotation**

The MQTT core server certificate expires, by default, in 7 days. You can set the expiration to any value between 7 and 30 days. 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. Existing connections are not affected. When the certificate expires, the AWS Greengrass core device attempts to connect to the Greengrass cloud service to obtain a new certificate. If the connection is successful, the AWS Greengrass core device downloads a new MQTT core server certificate and restarts the local MQTT service. At this point, all AWS IoT devices connected to the core are disconnected.

If there is no internet connection when the AWS Greengrass core attempts to get a new MQTT core server certificate, AWS IoT devices are unable to connect to the AWS Greengrass core until the connection to the Greengrass cloud service is restored and a new MQTT core server certificate can be downloaded.

When AWS IoT devices are disconnected from a core, they have to wait a short period of time and then attempt to reconnect to the AWS Greengrass core device.

**AWS Greengrass Cipher Suites**

AWS Greengrass uses the AWS IoT transport security model to encrypt communication with the cloud by using TLS cipher suites. In addition, AWS 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.

As opposed to the AWS IoT cloud, the AWS Greengrass core supports the following local network TLS cipher suites:

<table>
<thead>
<tr>
<th>TLS Version</th>
<th>Cipher</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLSv1.0</td>
<td>TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA</td>
</tr>
<tr>
<td></td>
<td>TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA</td>
</tr>
<tr>
<td></td>
<td>TLS_RSA_WITH_AES_128_CBC_SHA</td>
</tr>
<tr>
<td></td>
<td>TLS_RSA_WITH_AES_256_CBC_SHA</td>
</tr>
<tr>
<td>TLSv1.1</td>
<td>TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA</td>
</tr>
<tr>
<td></td>
<td>TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA</td>
</tr>
<tr>
<td></td>
<td>TLS_RSA_WITH_AES_128_CBC_SHA</td>
</tr>
<tr>
<td></td>
<td>TLS_RSA_WITH_AES_256_CBC_SHA</td>
</tr>
<tr>
<td>TLSv1.2</td>
<td>TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA</td>
</tr>
<tr>
<td></td>
<td>TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA</td>
</tr>
<tr>
<td>TLS Version</td>
<td>Cipher</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td></td>
<td><strong>TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384</strong></td>
</tr>
<tr>
<td></td>
<td><strong>TLS_RSA_WITH_AES_128_CBC_SHA</strong></td>
</tr>
<tr>
<td></td>
<td><strong>TLS_RSA_WITH_AES_128_GCM_SHA256</strong></td>
</tr>
<tr>
<td></td>
<td><strong>TLS_RSA_WITH_AES_256_CBC_SHA</strong></td>
</tr>
<tr>
<td></td>
<td><strong>TLS_RSA_WITH_AES_256_GCM_SHA384</strong></td>
</tr>
</tbody>
</table>
Monitoring with AWS Greengrass Logs

AWS Greengrass consists of the cloud service and the AWS Greengrass core software. The core software can write logs to CloudWatch and to the local file system of your core device. Logging is configured at the group level.

All AWS Greengrass log entries include a time stamp, log level, and information about the event.

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 Greengrass logs use the following naming conventions:

/aws/greengrass/GreengrassSystem/greengrass-system-component-name
/aws/greengrass/Lambda/aws-region/account-id/lambda-function-name

Under each log group, you see log streams with the following structure:

date/account-id/greengrass-group-id/name-of-core-that-generated-log

Be aware of the following considerations when using CloudWatch Logs:

- To enable logging to CloudWatch Logs, the following actions must be allowed in the AWS Greengrass group role:
  - logs:PutLogEvents
  - logs:CreateLogGroup
  - logs:CreateLogStream
  - logs:DescribeLogStreams
- Logs are sent to CloudWatch Logs with a limited number of retries in case there's no internet connectivity. After the retries are exhausted, the event is dropped.
- Transaction, memory, and other limitations apply. For more information, see the section called "Logging Limitations" (p. 195).

File System Logs

If you configure file system logging, the log files are stored under `greengrass-root/ggc/var/log` on the core device, with the following high-level directory structure:

```
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
```
Default Logging Configuration

If logging settings aren’t explicitly configured, AWS Greengrass uses the following default logging configuration after the first group deployment.

AWS Greengrass System Components

- Type - FileSystem
- Component - GreengrassSystem
- Level - INFO
- Space - 128 KB

User-defined Lambda Functions

- Type - FileSystem
- Component - Lambda
- Level - INFO
- Space - 128 KB

Configure Logging for AWS Greengrass

You can use the AWS IoT console or the AWS Greengrass APIs (p. 193) to configure AWS Greengrass logging.

Note
To allow AWS Greengrass to write logs to CloudWatch Logs, your group role must allow the required CloudWatch Logs actions (p. 191).
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 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 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:

```json
{
  "Loggers": [
    {
      "Id": "string",
      "Type": "FileSystem|AWSCloudWatch",
      "Component": "GreengrassSystem|Lambda",
      "Level": "DEBUG|INFO|WARN|ERROR|FATAL",
      "Space": "integer"
    },
    {
      "Id": "string",
      ...
    }
  ]
}
```

**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 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": "1",
        "Component": "GreengrassSystem",
        "Level": "ERROR",
        "Space": 10240,
        "Type": "FileSystem"
      },
      {
        "Id": "2",
        "Component": "Lambda",
        "Level": "INFO",
        "Space": 10240,
        "Type": "FileSystem"
      },
      {
        "Id": "3",
        "Component": "Lambda",
        "Level": "INFO",
        "Type": "AWSCloudWatch"
      }
    ]
  }
}
```

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 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 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 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.

\[ \text{greengrass-system-component-space} \times 8 \quad \text{// 7 if automatic IP detection is disabled} \]
\[ + 128\text{KB} \quad \text{// the internal log for the local logging component} \]
\[ + \text{lambda-space} \times \text{lambda-count} \quad \text{// different versions of a Lambda function are treated as one} \]

Where:

- `greengrass-system-component-space`
  - The maximum amount of local storage for the AWS Greengrass system component logs.
- `lambda-space`
  - The maximum amount of local storage for Lambda logs.
- `lambda-count`
  - The number of deployed Lambda functions.

Log Loss

If your AWS 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.
## Troubleshooting AWS Greengrass Applications

Use the following information to help troubleshoot issues in AWS Greengrass.

### Issues

<table>
<thead>
<tr>
<th>Symptom</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>You see 403 Forbidden error on deployment in the logs.</td>
<td>Make sure the policy of the AWS Greengrass core in the cloud includes &quot;greengrass:*&quot; as an allowed action.</td>
</tr>
<tr>
<td>Device’s shadow does not sync with the cloud.</td>
<td>Check that the AWS Greengrass core has permissions for &quot;iot:UpdateThingShadow&quot; and &quot;iot:GetThingShadow&quot; actions. Also see [Troubleshooting Shadow Synchronization Timeout Issues](p. 203).</td>
</tr>
<tr>
<td>The AWS Greengrass core software does not run on Raspberry Pi because user namespace is not enabled.</td>
<td>Run <code>rpi-update</code> to update. Raspbian has released a new kernel 4.9 that has user namespace enabled.</td>
</tr>
<tr>
<td>A ConcurrentDeployment error occurs when you run create-deployment for the first time.</td>
<td>A deployment might be in progress. You can run <code>get-deployment-history</code> to see if a deployment was created. If not, try creating the deployment again.</td>
</tr>
<tr>
<td>The AWS Greengrass core software does not start successfully.</td>
<td>• Check that you are using the binaries appropriate for your architecture.</td>
</tr>
<tr>
<td></td>
<td>• Check runtime.log for error messages. For more information, see [Troubleshooting with Logs](p. 199).</td>
</tr>
<tr>
<td></td>
<td>• Check that your AWS Greengrass core device has local storage available.</td>
</tr>
<tr>
<td>The AWS Greengrass core software does not start on a Raspberry Pi, and you receive the following error: Failed to invoke PutLogEvents on local CloudWatch, logGroup: /GreengrassSystem/connection_manager, error: RequestError: send request failed caused by: Post <a href="http://path/cloudwatch/logs/">http://path/cloudwatch/logs/</a>: dial tcp address: getsockopt: connection refused, response: { }</td>
<td>This can occur if the OS version is Raspbian Stretch. You can run the following command in the terminal to see the OS version that's installed on your Pi: <code>cat /etc/os-release</code></td>
</tr>
</tbody>
</table>

We recommend that you use the supported platform (p. 12) (currently Raspbian Jessie, 2017-03-02). However, if you choose to run the AWS Greengrass core software on Raspbian Stretch, add `cgroup_enable=memory` to your `/boot/cmdline.txt` file and then restart the device.
You might encounter other issues if you run AWS Greengrass core software on an unsupported platform.

The AWS Greengrass core software does not start, and you receive the following error: Unable to create server due to: failed to load group: chmod /greengrass-root/ggc/deployment/lambda/arn:aws:lambda:region:account-id:function:version/file-name: no such file or directory

If you deployed a Lambda executable (p. 126) 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 start Greengrass by running the following commands:

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, then you must point to the new function version or alias in your group definition and subscriptions before deploying the group.)

The AWS Greengrass core software does not start, and you receive the following error: Spool size should be at least 262144 bytes.

Open the group.json file (located in /greengrass-root/ggc/deployment/group), replace the contents of the file with an empty JSON object ({}), and start Greengrass by running the following commands:

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

Then follow the section called “To Cache Messages in Local Storage” (p. 26) procedure, and make sure to specify a GG_CONFIG_MAX_SIZE_BYTES value that's greater than or equal to 262144 for the GGCloudSpooler function.
<table>
<thead>
<tr>
<th>Symptom</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>The greengrassd script displays: unable to accept TCP connection.</td>
<td>The file descriptor limit for the AWS Greengrass core software has reached the threshold and must be increased.</td>
</tr>
<tr>
<td>accept tcp [::]:8000: accept4: too many open files.</td>
<td>Use the following command:</td>
</tr>
<tr>
<td></td>
<td>ulimit -n 2048</td>
</tr>
<tr>
<td></td>
<td>and restart the AWS Greengrass core software.</td>
</tr>
<tr>
<td>Note</td>
<td>In this example, the limit is increased to 2048. Choose a value appropriate for your use case.</td>
</tr>
<tr>
<td>You receive the following error:</td>
<td>Either install AWS Greengrass directly under the root directory, or ensure that the /greengrass directory and its parent directories have execute permissions for everyone.</td>
</tr>
<tr>
<td>Runtime execution error: unable to start lambda container.</td>
<td></td>
</tr>
<tr>
<td>container_linux.go:259: starting container process caused</td>
<td></td>
</tr>
<tr>
<td>&quot;process_linux.go:345: container init caused &quot;rootfs_linux.go:50: preparing rootfs caused &quot;&quot;permission denied&quot;&quot;&quot;</td>
<td></td>
</tr>
<tr>
<td>The deployment fails with the following error message:</td>
<td>Using the AWS CLI, check that an appropriate service role is associated with your account by using <code>GetServiceRoleForAccount</code> and ensure that the role has at least the <code>AWSGreengrassResourceAccessRolePolicy</code> permission applied. If you need to associate a Greengrass service role with your account, use <code>AssociateServiceRoleToAccount</code>.</td>
</tr>
<tr>
<td>Greengrass is not authorized to assume the Service Role associated with this account.</td>
<td></td>
</tr>
<tr>
<td>Symptom</td>
<td>Solution</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>The deployment doesn't finish.</td>
<td>Make sure that the AWS 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.</td>
</tr>
<tr>
<td></td>
<td>1. To check whether the daemon is running:</td>
</tr>
<tr>
<td></td>
<td>`ps aux</td>
</tr>
<tr>
<td></td>
<td>If the output contains a root entry for <code>/greengrass/ggc/packages/1.6.0/bin/daemon</code>, then the daemon is running.</td>
</tr>
<tr>
<td></td>
<td>The version in the path depends on the AWS Greengrass Core Software version that's installed on your core device.</td>
</tr>
<tr>
<td></td>
<td>2. To start the daemon:</td>
</tr>
<tr>
<td></td>
<td><code>cd /greengrass/ggc/core/; sudo ./greengrassd start</code></td>
</tr>
<tr>
<td>The deployment doesn't finish, and runtime.log contains multiple wait 1s for container to stop entries.</td>
<td>Restart the AWS Greengrass daemon by running the following commands in your core device terminal.</td>
</tr>
<tr>
<td></td>
<td><code>cd /greengrass/ggc/core/; sudo ./greengrassd stop; sudo ./greengrassd start</code></td>
</tr>
<tr>
<td>You receive the following error: Deployment guid of type NewDeployment for group guid failed error: Error while processing. group config is invalid: 112 or [119 0] don't have rw permission on the file: path</td>
<td>Ensure that the owner group of the path directory has read and write permissions to the directory.</td>
</tr>
</tbody>
</table>

If you're unable to find or resolve your issue using this information, you can search the AWS Greengrass Forum or post a new thread. Members of the AWS Greengrass team actively monitor the forum.

**Troubleshooting with Logs**

**GGC v1.6.0**

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 privileges.

`greengrass-root/ggc/var/log/crash.log`

Shows messages generated when an AWS Greengrass core crashes.
Troubleshooting with Logs

`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 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 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. 23) 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. Note that `crash.log` is found only in file system logs on the AWS Greengrass core device.

If AWS IoT is configured to write logs to CloudWatch, check those logs for information if connection errors occur when system components attempt to connect to AWS IoT.

For more information about AWS Greengrass logging, see *Monitoring* (p. 191).

GGC v1.5.0

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 privileges.

`greengrass-root/ggc/var/log/crash.log`

Shows messages generated when an AWS 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 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 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 the logs are configured to be stored on the cloud, use CloudWatch Logs to view log messages. Note that `crash.log` is found only in file system logs on the AWS Greengrass core device.

If AWS IoT is configured to write logs to CloudWatch, check those logs for information if connection errors occur when system components attempt to connect to AWS IoT.
For more information about AWS Greengrass logging, see Monitoring (p. 191).

GGC v1.3.0

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 privileges.

`greengrass-root/ggc/var/log/crash.log`

Shows messages generated when an AWS 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 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 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 the logs are configured to be stored on the cloud, use CloudWatch Logs to view log messages. Note that `crash.log` is found only in file system logs on the AWS Greengrass core device.

If AWS IoT is configured to write logs to CloudWatch, check those logs for information if connection errors occur when system components attempt to connect to AWS IoT.

For more information about AWS Greengrass logging, see Monitoring (p. 191).

GGC v1.1.0

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 privileges.

`greengrass-root/ggc/var/log/crash.log`

Shows messages generated when an AWS 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 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 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 the logs are configured to be stored on the cloud, use CloudWatch Logs to view log messages. Note that `crash.log` is found only in file system logs on the AWS Greengrass core device.

If AWS IoT is configured to write logs to CloudWatch, check those logs for information if connection errors occur when system components attempt to connect to AWS IoT.

For more information about AWS Greengrass logging, see Monitoring (p. 191).

GGC v1.0.0

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 privileges.

`greengrass-root/var/log/crash.log`
- Shows messages generated when an AWS Greengrass core crashes.

`greengrass-root/var/log/system/runtime.log`
- Shows messages about which component failed.

`greengrass-root/var/log/system/`
- Contains all logs from AWS Greengrass system components, such as the certificate manager and the connection manager. Using the messages in `var/log/system/` and `var/log/system/runtime.log`, you should be able to find out which error occurred in AWS Greengrass system components.

`greengrass-root/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 AWS Greengrass is configured to write logs to CloudWatch, you can view log messages in the CloudWatch console (p. 191). Note that `crash.log` is found only in file system logs on the AWS Greengrass core device.

If AWS IoT is configured to write logs to CloudWatch, check those logs for information if connection errors occur when system components attempt to connect to AWS IoT.

Troubleshooting Storage Issues

When the local file storage is full, some components might start failing:

- Local shadow updates do not occur.
- New AWS Greengrass core MQTT server certificates cannot be downloaded locally.
- Deployments fail.

You should always be aware of the amount of free space available locally. This can be calculated based on the sizes of deployed Lambda functions, the logging configuration (see Troubleshooting with Logs (p. 199)), and the number of shadows stored locally.
Troubleshooting Messages

All messages sent within AWS Greengrass are sent with QoS 0. By default, AWS Greengrass stores messages in an in-memory queue. Therefore, unprocessed messages are lost when the AWS Greengrass core restarts, for example after a group deployment or device reboot. However, you can configure AWS Greengrass (v1.6.0) 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. 25).

**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 Greengrass might not start properly.

Troubleshooting Shadow Synchronization Timeout Issues

GGC v1.6.0

If there is a significant delay in communication between a Greengrass core device and the cloud, then shadow synchronization may fail due to a timeout. You may see something like this in your log files:

```


[2017-07-20T10:01:58.006Z][ERROR]-sync_manager.go:375, Failed to execute sync operation {what_the_thing_is_named VersionDiscontinued []}"
```

A possible fix is to configure the amount of time your Greengrass core device waits for a host response. Open the `greengrass-root/config/config.json` file and add a `system.shadowSyncTimeout` field with a timeout value in seconds. For example:

```
{
    "coreThing": {
        "caPath": "root-ca.pem",
        "certPath": "cloud.pem.crt",
        "keyPath": "cloud.pem.key",
        "thingArn": "arn:aws:iot:us-west-2:049039099382:thing/GGTestGroup42_Core",
        "iotHost": "your-AWS-IoT-endpoint",
        "ggHost": "greengrass.iot.us-west-2.amazonaws.com",
        "keepAlive": 600
    },
    "runtime": {
        "cgroup": {
            "useSystemd": "yes"
        }
    },
    "system": {
        "shadowSyncTimeout": 10
    }
}
```
If no `shadowSyncTimeout` value is specified in the `config.json` file, the default is 1 second.

**GGC v1.5.0**

If there is a significant delay in communication between a Greengrass core device and the cloud, then shadow synchronization may fail due to a timeout. You may see something like this in your log files:

```
[ERROR]-cloud_shadow_client.go:57, Cloud shadow client error: unable to get cloud shadow what_the_thing_is_named for synchronization. Get https://1234567890abcd.iot.us-west-2.amazonaws.com:8443/things/what_the_thing_is_named/shadow: net/http: request canceled (Client.Timeout exceeded while awaiting headers)
```

```
[WARN]-sync_manager.go:263, Failed to get cloud copy: Get https://1234567890abcd.iot.us-west-2.amazonaws.com:8443/things/what_the_thing_is_named/shadow: net/http: request canceled (Client.Timeout exceeded while awaiting headers)
```

```
[ERROR]-sync_manager.go:375, Failed to execute sync operation {what_the_thing_is_named VersionDiscontinued []}
```

A possible fix is to configure the amount of time your Greengrass core device waits for a host response. Open the `greengrass-root/config/config.json` file and add a `system.shadowSyncTimeout` field with a timeout value in seconds. For example:

```json
{
  "coreThing": {
    "caPath": "root-ca.pem",
    "certPath": "cloud.pem.crt",
    "keyPath": "cloud.pem.key",
    "thingArn": "arn:aws:iot:us-west-2:049039099382:thing/GGTestGroup42_Core",
    "iotHost": "your-AWS-IoT-endpoint",
    "ggHost": "greengrass.iot.us-west-2.amazonaws.com",
    "keepAlive": 600
  },
  "runtime": {
    "cgroup": {
      "useSystemd": "yes"
    }
  },
  "system": {
    "shadowSyncTimeout": 10
  }
}
```

If no `shadowSyncTimeout` value is specified in the `config.json` file, the default is 1 second.

**GGC v1.3.0**

If there is a significant delay in communication between a Greengrass core device and the cloud, then shadow synchronization may fail due to a timeout. You may see something like this in your log files:

```
[ERROR]-cloud_shadow_client.go:57, Cloud shadow client error: unable to get cloud shadow what_the_thing_is_named for synchronization. Get https://1234567890abcd.iot.us-west-2.amazonaws.com:8443/things/what_the_thing_is_named/shadow: net/http: request canceled (Client.Timeout exceeded while awaiting headers)
```

```
[WARN]-sync_manager.go:263, Failed to get cloud copy: Get https://1234567890abcd.iot.us-west-2.amazonaws.com:8443/things/what_the_thing_is_named/shadow: net/http: request canceled (Client.Timeout exceeded while awaiting headers)
```

```
[ERROR]-sync_manager.go:375, Failed to execute sync operation {what_the_thing_is_named VersionDiscontinued []}
```

---

204
what_the_thing_is_named/shadow: net/http: request canceled (Client.Timeout exceeded while awaiting headers)
[2017-07-20T10:01:58.006Z][ERROR]-sync_manager.go:375,Failed to execute sync operation
{what_the_thing_is_named VersionDiscontinued []}"

A possible fix is to configure the amount of time your Greengrass core device waits for
a host response. Open the `greengrass-root/config/config.json` file and add a
system.shadowSyncTimeout field with a timeout value in seconds. For example:

```json
{
  "coreThing": {
    "caPath": "root-ca.pem",
    "certPath": "cloud pem.crt",
    "keyPath": "cloud pem.key",
    "thingArn": "arn:aws:iot:us-west-2:2049039099382:thing/GGTestGroup42_Core",
    "iotHost": "your-AWS-IoT-endpoint",
    "ggHost": "greengrass.iot.us-west-2.amazonaws.com",
    "keepAlive": 600
  },
  "runtime": {
    "cgroup": {
      "useSystemd": "yes"
    }
  },
  "system": {
    "shadowSyncTimeout": 10
  }
}
```

If no shadowSyncTimeout value is specified in the config.json file, the default is 1 second.

GGC v1.1.0

If there is a significant delay in communication between a Greengrass core device and the cloud,
then shadow synchronization may fail due to a timeout. You may see something like this in your log
files:

```plaintext
[2017-07-20T10:01:58.006Z][ERROR]-sync_manager.go:375,Failed to execute sync operation
{what_the_thing_is_named VersionDiscontinued []}"
```

A possible fix is to configure the amount of time your Greengrass core device waits for
a host response. Open the `greengrass-root/config/config.json` file and add a
system.shadowSyncTimeout field with a timeout value in seconds. For example:

```json
{
  "coreThing": {
    "caPath": "root-ca.pem",
    "certPath": "cloud pem.crt",
    "keyPath": "cloud pem.key",
    "thingArn": "arn:aws:iot:us-west-2:2049039099382:thing/GGTestGroup42_Core",
    "iotHost": "your-AWS-IoT-endpoint",
    "ggHost": "greengrass.iot.us-west-2.amazonaws.com",
    "keepAlive": 600
```

205
If no `shadowSyncTimeout` value is specified in the config.json file, the default is 1 second.

GGC v1.0.0

Not supported.
Document History for AWS Greengrass

The following table describes important changes to the AWS 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>AWS Greengrass Version 1.6.0 Released</td>
<td>New features: Lambda executables, configurable message queue, configurable reconnect retry interval, volume resources under /proc, and configurable write directory.</td>
<td>July 26, 2018</td>
</tr>
</tbody>
</table>

Earlier Updates

The following table describes important changes to the AWS Greengrass Developer Guide before July 2018.

<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
</table>
| AWS 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. 154).  
- Greengrass Lambda functions support binary input data, in addition to JSON.  
For more information, see AWS Greengrass core versions (p. 2). | March 29, 2018 |
| AWS 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 Greengrass Core Software (p. 114).  
- Access local peripherals and resources from Greengrass Lambda functions. For more information, see Access Local Resources with Lambda Functions (p. 129). | November 27, 2017 |
| AWS Greengrass Version 1.1.0 Released | New features:  
- Reset deployed AWS Greengrass groups. For more information, see Reset Deployments (p. 120).  
- Support for Node.js 6.10 and Java 8 Lambda runtimes, in addition to Python 2.7. | September 20, 2017 |
<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWS Greengrass</td>
<td>AWS Greengrass is generally available.</td>
<td>June 7, 2017</td>
</tr>
<tr>
<td>Version 1.0.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Released</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>